JONATHAN PEIRCE & MICHAEL MacASKILL

# BUILDING EXPERIMENTS IN *PsychoPy*

SAGE

Los Angeles | London | New Delhi
Singapore | Washington DC | Melbourne

# 6

# PROVIDING FEEDBACK: SIMPLE CODE COMPONENTS

**Learning objectives**: Here you'll learn how to add more flexible pieces of Python code to your experiment, with multi-line statements and the option to choose when they get executed.

The Builder interface is quite flexible in what it can achieve, but ultimately there's a limit to what can be done with such a graphical interface. When you want to go further, you can add custom Python code to almost any point in your study using 'Code Components'. With these, the options for your experiment become almost limitless. Most tasks need very little custom Python code but it's good to know the basics. The following examples all show you how to make use of such a Component.

As an example, we take the existing Stroop task from earlier and add a feedback option where the message can be customized according to the response of the participant. You might wonder why PsychoPy doesn't just provide some sort of feedback component so that you don't have to write code, but there are so many different ways you might want to customize the feedback you give that the dialog box to do this graphically would become as hard to use as writing the code! For instance, not only might you want to say 'correct' and 'incorrect', but you might want to tell people their reaction time (or not) or provide more complex criteria for what messages are given (if they get the answer right on this type of trial then do one thing, but if they get it right on another then do something else). The possibilities are limitless and this is where a code snippet can be so useful.

### Pro Tip: If you ask the forum for help then send your code

If you contact the PsychoPy users forum, asking for help because some aspect of your experiment doesn't work, then think carefully about where the problem might conceivably stem from. Hand-generated code is obviously the place where many experiments can go wrong. If you have any Code Components in your study then you probably need to copy and paste them into the forum post so that others can see your actual code if it might conceivably be relevant.

## 6.1 PROVIDING FEEDBACK

One very common requirement is that we need to provide feedback to a participant. In this case we need an `if` statement so that `if` they are wrong the text of a feedback object says 'You were wrong' and gives them some positive message if they were correct. This can't easily be inserted into a Builder Component. What we can do in this case is insert a Code Component.

Open your Stroop experiment from Chapter 2. Now the first thing you need to do is add a new Routine called `feedback` and place it immediately after the `trial` Routine, but within the same loop. Click on the `Insert Routine` button in the Flow, as in Figure 6.1. Select the `(new)` option and give your Routine the name `feedback`.

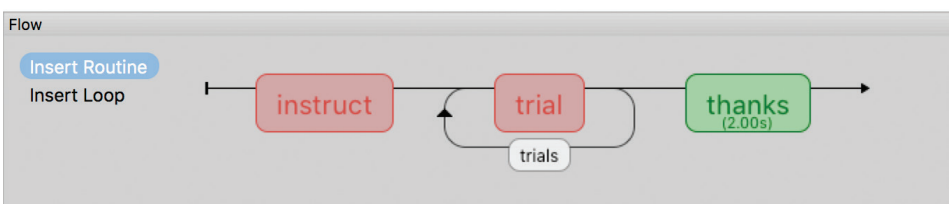


**FIGURE 6.1** Inserting a new Routine into the Flow of our Stroop task.

When you've typed in the name you can select the place where the new Routine will be inserted. Put it after the `trial` Routine but before the end of the loop. Your flow should now look like Figure 6.2.

Now select the `feedback` Routine (either by clicking on it in the Flow or by selecting its tab in the Routines panel). It should be an empty Routine with no Components. We need to use two Components here, a Text Component to

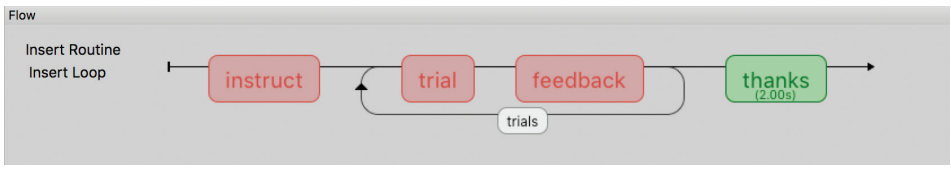


**FIGURE 6.2** Flow of the Stroop task after adding the `feedback` Routine.

draw the actual feedback text and a Code Component to determine the contents of the feedback message. The order they appear in is important: the code within the Routine is always run in the order it appears, with the top item being run first. We need to make sure that the Code Component is executed before the Text Component or the contents of the Text Component will not be set as expected. We can reorder Components afterwards by right-clicking on them and moving up/down, but it's easier just to create them in the correct order (code first, in this case).

Create a Code Component and give it the name `setMsg` to remind yourself that this code is setting the contents of the feedback message. Now, a Code Component has several tabs to set code that will run at different points in the experiment. Think carefully about what your code needs to do and therefore at which locations it needs to run. For some things you may need to execute code in multiple parts of your study. For instance, with an eye tracker you

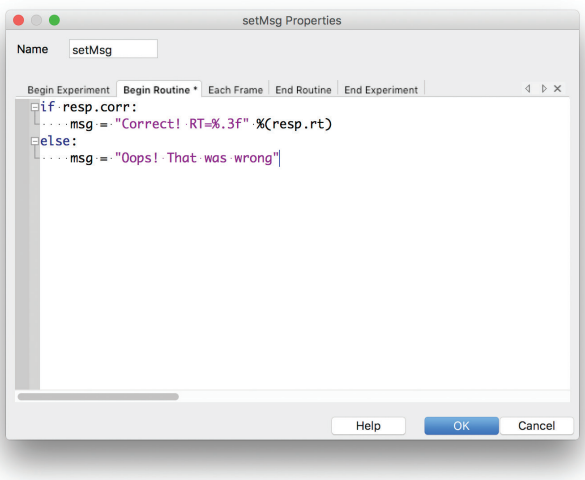


**FIGURE 6.3** Setting the properties of the `setMsg` Code Component. All this is set to occur at the beginning of the Routine.

may need to initialize the hardware at the start of the experiment, then reset its clock and start sampling at the start of the Routine, check the eye position on each screen refresh and then disconnect the hardware at the end of the study. In our study all we need is some code that runs at the start of the Routine. Note that this is running immediately after the `trial` Routine and any variables created there will still be available to us in our new `feedback` Routine. Insert the code in Figure 6.3 into the tab called `Begin Routine`. Make sure you type it exactly. Note that two of the lines are indented (you can press `Tab` to get the indent).

### Pro Tip: Choose good names for Components

It's a really good idea to give a little thought to the names of the Components that you add to your experiment. While you're adding your Component its purpose was probably (hopefully) obvious, but when you come back and look at the experiment later on (next year when you create a related study) the contents of each Component will not be obvious at all. It will then be really annoying if you have to click on each item to see what it was because the names they were given were simply 'text_1' or 'code_3'. Also, any components that are going to appear in the data file (e.g. keyboard items) need to be given good names so that in the data files you can see what they were: Was this the keyboard that merely advanced the experiment or was it the one where the participant gave their response?

Our Code Component creates the variable called `msg` at the beginning of the Routine and sets it to one of two values. The values are based on `resp.corr`. This only works because our Keyboard Component in the `trial` Routine was called `resp` and it always has an attribute called `corr`. Note that your data files also have a column called `resp.corr` if you've set them up in this way. The value of `resp.corr` is either `True` or `False` (or 1 or 0, which are equivalent to `True` or `False` in Python) and if `resp.corr` is `True` then we set `msg` to be the text 'Correct!'

So now all we need to do is create our Text Component to present our feedback message. The variable, `msg`, can be used just like the variables we were fetching earlier on from our conditions files, by inserting it into Components using the `$` sign to indicate code. Go ahead and create a new Text Component and give it the values in Figure 6.4. Remember to set your message text to update `every repeat`.

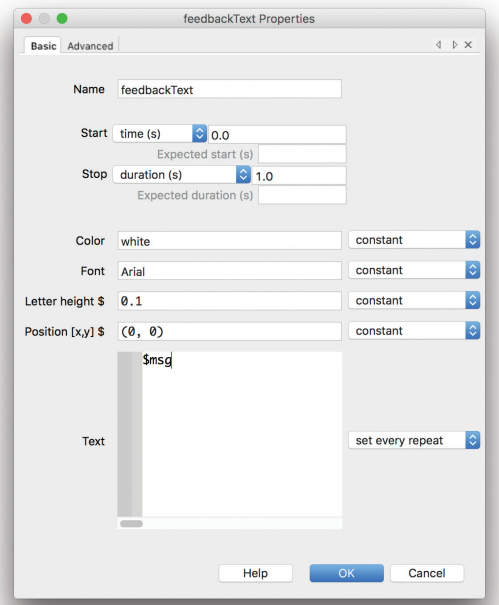


**FIGURE 6.4** Properties of the `feedbackText` Component. Note that '`$msg`' is going to be '`set every repeat`' and that this needs to be done after the variable '`msg`' has been created (in your Code Component).

### Warning: Order of Components in a Routine

Code for each Component of your Routine is executed in the order that the Components appear on the screen. For stimuli you can use this order to decide which stimulus appears 'in the front'. Lower stimuli in the Routine are executed later, so they will obscure stimuli that are higher in the Routine. In the case of a Code Component you usually want the code to come before the stimuli that it will affect (otherwise it will affect them on the *subsequent* frame or the *subsequent* routine not the current one).

## 6.2 UPDATING THE FEEDBACK COLOR

This already works (or, if not, go back and make sure your text matches exactly what was in Figure 6.4) but we could tweak it further. Let's set the text to be different colors as well. We could set it to be red if the participant was wrong and green if they were right. To do that we need to add a couple of extra lines into

our `if...else` statement. As well as creating the variable `msg` we create `msgColor` and set it to either 'red' or 'green':

```
if resp.corr:
   msg = "Correct!"
   msgColor = "green"
else:
   msg = "Oops! That was wrong"
   msgColor = "red"
```

Now you need to go to the Text Component again and alter its properties so that the color is set to be `$msgColor` and the updates are set to `every repeat` for this as well.

## 6.3 REPORTING THE REACTION TIME

We could also set the feedback message to report the reaction time (let's just do this if the participant gets the answer correct). This is the most complex step of the current chapter so pay attention! The reaction time from a keyboard response is stored as an attribute `rt` and can be accessed using the variable `resp.rt`, where `resp` is the name of our Keyboard Component. So how do we insert the value of this reaction time (which is a number) into our string of text? In Python we can convert a number to a string using the function `str()` and we can add two strings together just using `+`, so the following would work:

```
msg = "Correct! RT=" + str(resp.rt)
```

Although this works, there are two problems. The first is that the reaction time is displayed to a ridiculous number of decimal places. Although the computer clock records the time with a precision of approximately nanoseconds, the reality is that your keyboard is not that precise. Let's reduce the number of decimal places to 3 (giving the reaction time effectively in milliseconds). For a standard USB keyboard this is still more precise than is warranted, but more reasonable. The second is that, if we wanted to add other text or other variables to this message, the method of *adding* strings quickly gets cumbersome. An alternative is to use Python string formatting. There are several different systems for doing this that end up being roughly equivalent, but we'll look at the newer one using `"{}".format(value)` which will be familiar to people used to programming in `.NET`. There is yet another, older method, using insertions like `%i`, `%f` that you may be more comfortable with if you know those methods from C or MATLAB

(see https://pyformat.info/ for more detail about both methods) but we won't get into that method as well!

```
msg = "Correct! RT={:.3f}".format(resp.rt)
```

The way this works is that in the string there are some curly brackets `{}`, which indicate to Python that a variable needs to be inserted here. Then the characters `:.3f` indicate that it should be formatted as a floating point value with three digits after the decimal place. After the string finished it was 'formatted' by the function called `format()` to which we gave a variable to be inserted `resp.rt` (it could have been several variables but we only needed one). So we inserted `resp.rt` as a floating point value correct to 3 decimal places `{:.3f}` in our string.

We could easily have added further text after the insertion, or altered the value of the float or added other variables to the feedback as well. For instance, we could have:

```
msg = "{} was correct! RT={:.3f} ms".format(resp.keys, resp.rt*1000)
```

The code above inserts two variables, one of which is a string (`{}` fetches the value `resp.keys`) and one of which is an integer (`{:i}` with value `resp.rt*1000`). You can see there is further text after the final integer giving the units ('ms').

## Pro Tip: String formatting options

Here are some of the common ways of formatting inserted variables using string `format()` functions.

| Code | Effect |
|---|---|
| `{}` | Auto-insert with default formatting |
| `{:s}` | Insert a string |
| `{:.5s}` | Insert first 5 characters |
| `{:f}` | Insert a float |
| `{:.2f}` | Insert a float with 2 decimal places |
| `{:i}` | Insert an integer |
| `{:03i}` | Insert an integer padded with zeros to be 3 characters long (001, 002, . . . ) |
| `{:3i}` | Insert an integer padded with *spaces* to be 3 characters long |
| `{:+i}` | Insert an integer but force it to show its sign |

The syntax above is based on a system in Microsoft's `.NET` Framework.

The website https://pyformat.info/ provides a fantastic reference for the myriad ways to format your strings using either syntax.

## 6.4 IDEAS FOR USEFUL CODE SNIPPETS

Table 6.1 should give you some ideas of pieces of code that you could add to your experiment, but there are infinitely (or at least a lot!) more options. How do you find them? How, for example, could you possibly have deduced that `t` was the value for the current time in the Routine, or that `random()` gives you a random number (rather than `rand()`, for example)? The best way to see what things are possible is to compile your experiment into a script and use that to go and investigate the code that will be in place. Any of the objects you see in that script you could also use yourself, in your Code Components or in the parameter boxes of your components.

**TABLE 6.1** Some useful code ideas to insert into your Builder experiments.

| Code | Effect |
|---|---|
| `random()` | Inserts a random number. Varies between 0 and 1 |
| `t` | Time (in seconds) since start of Routine |
| `frameN` | Number of (completed) frames since start of Routine. Note that this is 0 on first frame and 1 on the second frame etc |
| `frameDur` | The (expected) duration of each frame. This is measured at the start of the experiment, if possible, and set to 1/60.0 if not |
| `if frameN % 5 == 0: .....` | Do something every 5 frames. Google for 'modulo' to find out about %. Don't insert the dots! These are just our way of saying 'put your code here'! |
| `globalClock.getTime()` | Time since start of study. Note that this is not the start of the first trial; it will include a period at the beginning where stimuli were loading and the dialog box was waiting for input |
| `continueRoutine=False` | Ends the *current* Routine. This cannot be applied to arbitrary Routines, unfortunately, only to the current one |
| `trials.finished=True` | Ends a loop (here we used the `trials` loop). This ends at the point when the loop goes to its next iteration, which may not be immediately |
| `expInfo['participant']` | Access values from the info dialog box (here we fetch the `participant`). This is a very useful way of controlling, for instance, which block someone is in, by creating a variable in the info dialog |
| `stim.status==FINISHED` | Tests whether the `stim` component has finished. Often used to sync two stimuli. Possible values for status are `FINISHED`, `STARTED` and `NOT_STARTED`. Note that a single '=' will usually give a syntax error here |

### Pro Tip: Finding out what attributes an object has

Table 6.1 gives you some idea of the sort of attributes that PsychoPy objects have, but to make it exhaustive would require a bigger book than this one! So how would you work out what attributes an object in a Python script has? Compiling the experiment to a script and seeing what is commonly used is one way, but most objects have more attributes that aren't used in every script, so how would you find those?

Python has a function called `dir()` that you can apply to anything to find out what attributes it might have. For instance, to find out what values exist in the `trials` loop you could do `print(dir(trials))`. You need to do this at some point after it has started. You could create a Code Component to execute it at the beginning of a Routine (choose one that's inside the loop so that the loop will have been created).

This will output many different items, including the following:

```
'nRemaining', 'nReps', 'nTotal', 'thisIndex', 'thisN',
'thisRepN'.
```

These indicate that items like `trials.nRemaining` and `trials.nReps` are variables that you could access within your scripts. This trick of using `dir()` works with almost anything.

## 6.5 REPORTING PERFORMANCE OVER THE LAST FIVE TRIALS

Often people want to keep track of the performance of participants, particularly as a criterion to exit a practice period (see Exercise 6.2). We'll measure performance over the *last five* trials and report it back to participants. Actually, we could use a larger period, like the last 10 trials, but the key is that we don't simply take the overall average; we want something that focuses more on recent performance rather than overall performance.

Save a new copy of the Extended Stroop demo but, instead of reporting the performance of the last trial to participants, take the average over the last five trials. To do this you'll need to adapt your current feedback code.

To track performance you *could* interrogate the Experiment Handler to fetch the data (which it stores for you as you go) but that requires more knowledge of the underlying objects. An easier solution is to keep your own list of whether or not participants were correct, and update and check this after each trial.

We can do this just by making a few changes to the Code Component called `message` in the `feedback` Routine that was already being used.

Open up that `message` Component and set the `Begin Experiment` tab to the following. The line creating the (empty) `msg` was probably there already. Thus:

```
msg = ''
corr = []
```

This just creates an empty Python 'list' object which allows us to store an arbitrary set of things and add to it along the way. What we need to add is, following each trial, whether or not the participant got the correct answer. From the existing code you should be able to spot that this is simply `resp.corr`. To add this to our list object called `corr` we need to use the `append` method, like this:

```
corr.append(resp.corr)
```

OK, so now we're keeping track of the sequence of correct/incorrect responses. Lastly, we need to use this list to calculate a performance. How would we do that in Python? If we keep adding entries to the list, one on each trial, then we want to extract the final five items in that list. In Python you can extract particular parts of a list (or letters in a string) by using square brackets. The general shape of those queries is `variableName[start:end]` (and since Python starts counting at 0) so:

- `corr[0:5]` would be the *first five* entries
- `corr[3:8]` would be five entries starting from the fourth (because we start counting at 0, number 3 is the fourth number!)

If you don't specify the start/end points (on either side of the `:`) then Python will take the entries from the *very start* or to the *very end*. Also, if you specify negative values as indices then it will represent the number backwards from the end of the list rather than the number forwards from the start. For example:

- `corr[:5]`, as before, gives the first five entries
- `corr[3:]` gives all the entries, starting from the fourth
- `corr[-5:]` gives the last five entries

Now we know how to extract the last five entries, we need to know how many correct answers there were. Given that we store these as 1 for 'correct' and 0 for 'incorrect' we can simply use the Python function `sum()` to calculate the total correct from a given list (or subset). Let's create a variable called `nCorr` that takes the sum of the last five.

We can keep adding this to the same place (the `Begin Routine` section of the `message` Code Component). This section should end up looking like this:

```
# create msg according to whether resp.corr is 1 or 0
if resp.corr: # stored on last run of routine
```

```
    msg = "Correct! RT={:.3f}".format(resp)
else:
    msg = "Oops! That was wrong"

# track (up to) the last 5 trials
corr.append(resp.corr)
nCorr = sum(corr[-5:])
nResps = len(corr[-5:])
msg += "({} out of {} correct)".format(nCorr, nResps)
```

The first lines of the code create the conditional `msg`, as before. Then we append the value `resp.corr` to our list of responses and calculate the number that were correct in the last five. We also want to know *how many* trials that is taken over; it would be strange to tell the participant '1 out of 5 correct' if they've only actually had two trials. The code `len(corr[-5:])` tells us the *length* of the list subset that we're using, just as `sum(corr[-5:])` tells us its sum.

The last step is to append this to the value `msg` that we've already created. You could use `msg = msg + .....` but we used the special shortcut `+=` to do that for us. This is a common shorthand for 'add something to my variable' used in many programming languages.

### Pro Tips: Inserting good comments

Note that, even though our code block only has a few lines in it, we inserted several comments. That isn't just because this is a textbook, but something you should keep doing to remember how your code works. **Insert lots of comments for yourself.** You'd be amazed how, in the future, you won't remember what the lines of code were for. You can think of comments in your code as a love letter from current you to future you. Future you is probably very busy, so be nice!

## 6.6 EXERCISES AND EXTENSIONS

### EXERCISE 6.1: ABORT PRACTICE WHEN PERFORMANCE REACHES 4/5 CORRECT

What happens when you want to abort your practice session based on a performance criterion? In Section 6.5 you learned how to track what the

*(Continued)*

performance was over the last five trials. Try and use the calculations in that code to abort your practice trials when participants reach 4/5 correct.

Solution: Page 288

### EXERCISE 6.2: SHOW PROGRESS ON THE SCREEN WITH TRIAL NUMBER

Sometimes you want to be able to show progress through the experiment. Sometimes that's for the participant's morale in a long and boring study, and sometimes it helps you debug an experiment to see what's going on. There are also various ways to do it (e.g. in text or in a progress bar).

Insert a Text Component into your experiment, positioned in the lower right corner, that gives the number of completed trials in this loop and the total number of trials to be conducted, for example a simple string like '4/30'. This could be done for any of your experiments so far (e.g. just the basic Stroop task).

To do this you need to know how to identify the right numbers from the `trials` loop and combine them into a string.

Solution: Page 288