# Editing

In general, initial entry for code in LillyPy is the same as for plain text. The difference is that as you type, you are creating larger units of text (and in some cases graphics) that henceforth become an individually manipulable object

## Parens are Weird

While LillyPy representation may look a lot like text, it’s really hierarchical, like the underlying language itself. The place that will look the weirdest at first is when you’re typing parentheses, brackets and braces. You won’t see much difference when you’re typing a new expression, but when you start editing existing text, the differences will start to appear.

### Different Kinds of Parens

One difference that you’ll see immediately when you start using LillyPy; is that parentheses, brackets, and braces, do not have the same uniform appearance that they do in text. There are multiple kinds of each, and while you type them with the same keys, once typed, they each have different meanings. Unlike syntax highlighting in an IDE, they indicate an identity that generally persists across edits.

Parentheses:

1. Arithmetic auto-parens – Expressions in LillyPy, while they may look like plain text, are actually a hierarchy of operations. In a text editor, if you type “3 \* ” and then paste “2 + 2” after the “\*”, you’ll get “3 \* 2 + 2”. In LillyPy, you’ll get “3 \* (2 + 2)”. The parentheses are actually part of the “+” operation. If you then grabbed the “(2 + 2)” back out of that expression and pasted it after “3 + “, the parens would disappear and you would get “3 + 2 + 2”.
2. Tuple parens – If you type a comma inside of a paren, you will get a tuple.
3. Call parens – If you type a paren after an identifier, LillyPy will produce an italic-looking paren referred to as a call paren.
4. Grouping parens – Used for emphasis but not syntactically necessary, around arithmetic grouping that already conforms to expression rules for precedence and associativity.

Brackets:

1. Lists brackets – Surround a list or list comprehension
2. Subscript brackets – Surround an array reference or slice

Braces:

1. Dict/Set – Surround a dictionary or set constant

Constructive Parens/Brackets/Braces

1. Temporary open paren/bracket/brace to help type one of the above types.

LillyPy inserts a “constructive” paren, bracket or brace when it is not yet sure where the matching end will go, or in some cases, what kind of paren you will be typing. Until these are closed, it’s best to think of them as placeholders, as opposed to “real” parens, because they don’t balance like normal parens. For example, if you typed fn(1\*2 +3), and then inserted an open paren before the 3, you would see:

fn(1\*(2+3)

In a text editor, you could type an end paren at the end of the expression to balance it. In LillyPy, you could do that if all the parens were of the normal arithmetic/grouping type. However, here they are part of a function call, which are not interchangeable.

Another aspect of parenthesis to you will need to become accustomed, is building them left-to-right. None of these restrictions should lead to additional keystrokes or mouse movementsThe LillyPy representation is hierarchical to is need to type it after the 3, because. In LillyPy, cursor parens don’t count until they are closed, and you can’t close it by typing an end paren at the end of the expression

(It would be possible to allow this, and probably worth doing. That is, make the rule that the cursor paren can cause rematching of all of the parens in the expression between it and the matching paren. This would only hold for arithmetic parens, not braces, brackets, calls, or tuples.)

The rule of thumb with parens is to type them left to right. If you want to enclose something, type the left one first, then the right. To save effort, you don’t actually need to move the cursor. You can also use the ^) key to toggle between all of the possible matching end positions). Likewise, deleting the right paren re-opens the paren, whereas deleting the right paren removes both.

Backspacing has a different effect on an end paren/bracket/brace than it does on a start-paren.

## Fragments

Files representing python modules have a sequence attachment at the top which represents the code executed when the module is imported. You may have noticed, however, that it is also possible to place code randomly, anywhere in the window that is not attached to that main sequence. These fragments are ignored during module import, and it’s generally bad form to leave them in place in finished code. Fragments are a unique capability of LillyPy, which allow you to construct and test code in close proximity to the code under development. It may take a while to get accustomed to using them as they are so different from what was ever possible in conventional text coding environments.

# Interactive Use

## Mutable Objects

When you type the python bracket syntax for a list, for example [2,1], you are creating an expression to create a list. Every time you execute that expression, you will create a new list. LillyPy also allows you to directly edit mutable data in a way that actually changes the underlying data. If you type [2,1] in a LillyPy window and then execute it, you will get back a similar list [2,1], but with a slight difference: a small circle near the top of the bracket. A small circle on a bracket or brace indicates that it represents underlying mutable data, as opposed to just an expression for creating that data. In the above example, this distinction is entirely unimportant, because the data is not visible or in use in any other place. However, if you were to add a .sort() and execute it, you would see that the list suddenly rearranges itself from [2,1] to [1,2]. This capability to modify mutable data, allows you to work with data and intervene in the execution of code in ways that have been difficult in the past. In particular, it allows you to more easily work with code fragments and experimentally execute code before it is fully conceived and written, doing some steps by hand and some with code.

## Modifying Data

When you type/paste/drag data anywhere within the tree beneath mutable data, you are making modifications to the mutable data. Some of these modifications will take effect immediately and automatically, for example dragging a new value in to a list. Other changes require you to execute to make the change, for example if you drag/paste/type an expression that needs to be evaluated, such as 3 + 3.

Often, you will see a red dot appear in place of the circle-indicator on a mutable icon. The red dot means that you have edits pending. If you enter an expression or some other (not purely data) construct that requires you to manually execute to enter the data, it will come on and stay on until you execute or undo what you have typed. It will also come on as you start typing, and then either go out as you finish if the item you typed was purely-data, or stay on if you typed something that requires execution.

Often when working with mutable data, the normal Execute (double-click or Crtl+Enter) command, is overkill, because it executes everything and plops an extraneous result in to the window. A better command for updating just the mutable data in your expression, is “Update” (^U). Update executes and replaces just the data within mutable objects, without returning a value.

Because live mutable data can be changed both by user-edits and by execution of unconnected code, there can be cases where these conflict. Normally, mutable icons update automatically to track changes made to them by executing code. However, these updates are not applied when there are pending edits (the red indicator on the object is lit), so you will stop seeing changes being made externally to the object, and your edits, when complete, will overwrite all such changes. Two particular cases are worth noting. One is related to Undo, which is discussed separately, below. The other is what happens when a modification happens externally to mutable data which also has a pending edit. Normally if you back out of a pending edit, either via Undo or simply by backspacing out what you’ve typed, the icon will resume automatic updates to reflect external changes. However if an external change happened while an edit was pending, and then that pending edit was cancelled, the icon will stay in the “pending edit” state with the red indicator lit, until you manually resynchronize it (menu item) with the underlying data. (… I think a command may also be needed to revert mutable icons to match the underlying data)

The interaction between execution and Undo in this system may seem strange, at first. If you execute some code or data, and then type Ctrl+z to undo, the data result that the execution created will go away, just as if you had actually “undone” the execution. However, you’re only really undoing what was displayed *as a result of* execution; the underlying data objects remain unchanged (red dots will appear on mutable objects to indicate that they are out of sync with what is displayed). Should you indeed want to undo a change to a mutable object, you can follow the Undo command with an execution or Update (^U) command to update the corresponding data object to match what is displayed.

## Mutable Data as Assignment Target

In general, you can use mutable data anywhere you could use either its code counterpart (same icon without the mutable marker), or a variable pointing to it. For example, unlike its code counterpart, a mutable icon can be an assignment target (the code form of a list can appear on the left side of an assignment for the purpose of grouping, but is not itself a target). As you might expect, you can write [1, 2][0] = 3 (where [1,2] is a mutable list acting like a variable). As an additional special case, direct assignment to a mutable list is also possible: [1,2] = [3, 4, 5] means “replace the *content* of the mutable list with whatever is on the right side of the =” (currently lists are the only mutable data type supported for direct assignment).

## Immutable Data Objects

For the Immutable data objects, such as constants and tuples, there is no visible difference between an icon you type and one that is returned from executing code, and no reason to treat them differently.

Under the hood, there can be a subtle difference that will only matter in unusual circumstances. This difference relates to Python’s “is” comparison. While doing comparisons against the identity of an object rather than its value is usually discouraged, there are occasionally valid reasons to do so, as well as reasons that it gets done by accident and remains undetected in working code. To ensure that you can safely edit live data from code that cares about object identity, LillyPy takes measures to make sure that data icons representing larger compound immutable data objects retain the identity of the data they represent. While you won’t see any difference between an icon you type and one that is returned via code execution, the one you pull out of output data will consistently point to the original object when executed. Of course, as with other features of live data, this identity is not preserved across saves.

## Other Limits of Live Data

When you’re editing live mutable data, there are some editing operations that you can do that will cause it to lose its “liveness”. Most of these are obvious, such as adding comprehension code to a list, but others are less so, such as converting a dictionary to a set. Any edit to immutable data (a tuple) will, of course, break its link with the data object it may have represented, since the only way to modify immutable data is to replace it.

Just as you can build impossible data topologies with code, it is also possible to do so by manipulating data icons by hand. For example, if you have two mutable-list icons that both represent the same list, it is possible to drag one of them inside the other. LillyPy will detect that particular issue and convert the inner list to a list-definition (normal bracket without circle). However, it is still possible to fool it in to creating something circular, particularly if code execution is involved. If a bad data edit makes it past LillyPy’s checks, it will usually appear as an execution error, since even data edits are normally handled via execution.

One minor point of confusion is that there is currently no distinguishing the mutable data icon for the empty set from the mutable data icon for an empty dict. (… sets should probably have a different style of curly brace, but currently don’t).

## Save File Format

The format used for save (.pyg) files and for clipboard transfer of icons between LillyPy windows, is Python code in text form, with a simple macro overlay. The macro annotation is used to attach attributes, such as layout hints, and to implement extensions to the Python syntax. .pyg files are readable/editable text files that are compatible with version control systems and other legacy software tools.

To convert .pyg files back to standard Python, use the File->Export command. Pasting code copied from LillyPy in to a text editor, will also yield standard Python. In the very worst case, should you somehow lose all access to the LillyPy environment, the .pyg format is also simple enough that most code can be transformed back just by stripping the macros with a few lines of Python.

# Making Code Graphical and Interactive

Making a library with graphical and interactive features is of course more work than writing a conventional programmatic interface. While someday this may be common for all types of software, the users with the most immediate need are those who already do interactive computation and particularly those who blend it with visualization.

## The .pgi File

To add interactive capabilities, libraries for the LillyPy environment provide both a .py (or .pyg or .pyc) file containing the callable code for the library, and a .pgi file defining the interactive part. Whereas the imported .py file is not referenced until the user runs the code that imports it, the .pgi code is immediately loaded in to the environment and invoked during editing. It provides both traditional GUI components like menus and dialogs, graphics to embed directly into the code representation.

## Making an Icon

While LillyPy’s representation looks a lot like text, under the hood, it’s really made of icons (you can see the outlines when you drag). Icons give LillyPy the context to understand what the user is pointing at. Most of LillyPy’s graphical features are tied to them in one way or another.

The simplest sort of icon is defined only in terms of text, with maybe a font or color to distinguish it. A text icon can still have attachment sites and support context sensitive menus. On the other end of the spectrum are fully graphical icons that control the layout of subordinate icons.

### Macro Language

One of the jobs of an icon is to generate text that will represent it in the .pyg file and on the clipboard. The .pyg file looks like Python with some additional information overlaid in the form of macros.

While LillyPy’s macro language can do simple lexical substitution, its main purpose is to direct the icon generation that happens after the Python parser has parsed the file in to an abstract syntax tree. It’s most important functions, therefore are:

1. Tagging AST nodes with annotation, such as layout hints
2. Associating icon-creation functions with the AST nodes on which they will operate

The save-file format extends Python with macros of the form:

$name:arg-string$

name is composed of the same set of characters as Python identifiers. Macros that skip the name ($:args$) provide (mostly layout-related) information to the built-in icon creation functions for Python itself. The colon separates the macro name from its arguments, and may be omitted if there are no arguments to pass. The format of the argument string (argString) is entirely up to the implementer, but must not contain the "$" character. Defining a macro is the first step toward creating an icon. To define a macro in your .pgi file, import the lillypy package and call:

lillypy.addMacro(name, subs, iconCreateFn)

name defines the name that will reference the macro. subs, provides text to replace the macro before parsing with the Python parser, and may alternatively be passed as a function to generate the substitution string from the macro arg-string. Since the ultimate goal is to create icons, string substitution is needed only in rare cases to temporarily support sub-structure (such as statement blocks) and get it to pass initial parsing. Most of the work will be done in the icon creation function (iconCreateFn)

A macro ultimately associates its argument string and its icon creation function with a node in the abstract syntax tree (AST) created by the Python parser. To do that, it must be placed precisely in the .pyg file, before the Python construct to which it should be attached (or in the case of a macro that does string substitution, a $ character should be placed before the construct in the inserted text).

### Icon Creation