# Editing

In general, initial entry for code in LilyPy 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 . We call these objects icons, even though most of the time they just look like text and have no visible borders. The points where icons connect are called sites. Icon borders and sites do appear when you drag them around, but disappear once you’re done so as not to detract from the appearance of the code.

There are five standard site types and a few specialty ones that will come up later. The standard ones are expression sites, attribute sites, sequence sites and annotation sites. (… more description)

Like a text editor, LillyPy has a blinking insertion cursor, where text will go when you type. Unlike a text editor, the cursor travels between icon sites and takes on different forms depending on the type of site.

## Entry Icon

Most icons map directly to a syntactic element of the programming language (identifier, operator, etc.), but on the way to becoming one of these syntactic elements, the text you type starts as undifferentiated plain text. When you start typing, the editor will pop up a small text-box called an “entry icon”, showing the characters you type, until they are fully formed in to a unit of language syntax. If you click away from the text box and the contained text cannot be converted to a language element, it will remain in placeholder form, ready for you to continue entry by clicking back in the text to place the cursor.

### Pending Arguments

The entry icon has a second function that you won’t see when you’re initially constructing code, only when you go back and edit the code you’ve already entered. That is the function of holding on to code that is temporarily orphaned as you enter new code around it. A program is basically a tree structure, and editing that tree structure often means inserting and deleting items between other items. This requires temporarily disconnecting one or more branches from the tree structure and joining them back, later. While this sounds complicated, for the most part you type almost exactly what you would in a text editor. What you see is also similar, except for some additional coloring to remind you what still needs to be reconnected with the surrounding code.

There are a few cases where deletion will orphan more than a single branch of the code tree. While you might expect this to happen a lot, Python syntax accepts lists in almost any context so there is almost always a syntactically correct place to attach liberated code, and you are rarely left with more than a single unattached fragment. In the few cases where you are, the entry icon will hang on to the unattached code until you type something that can use some or all of the additional arguments. If it can use some but not all, it will use the pending arguments up to the one that it can’t, and create a new entry icon to hold the remaining one(s) .

## Parens are Weird

While LilyPy 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 entering new text, but when you start making edits, the differences will start to appear.

### Different Kinds of Parens

One difference that you’ll see immediately when you start using LilyPy; 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 actually indicate an underlying structure.

Parentheses:

1. Arithmetic auto-parens – Expressions in LilyPy, 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 LilyPy, 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, LilyPy 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. List 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.

LilyPy 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. For example, if you typed fn(1\*2 +3), and then inserted an open paren before the 2, 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 LilyPy, 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.

You can think of constructive parens as placeholders, but they are more than that. They affect the hierarchy of the code that follows them. In fact, an un-closed paren encloses its entire *potential* scope, which means it has more effect on structure than a closed one.

Another aspect of parenthesis to which you will need to become accustomed, is that LilyPy treats insertion and deletion differently depending upon which end of the paren/bracket/brace pair you are operating on.

While you are still learning, and treating LilyPy like a text editor, you can make it behave more like one by following the rule of thumb of always typing parens left to right. For example, if you want to enclose something in square brackets, move the cursor to the where the left bracket should go, type the left bracket first, move the cursor to where the right bracket should go and type the right bracket. If you start to get annoyed that you can’t do it the other way around (“my cursor is nearer the end bracket!”), you have reached a teaching moment, and it’s time to realize that this is not a text editor. You don’t need to move the cursor to both ends to place a paren, bracket, or brace. If you cursor is nearer the end paren location, move it there and use CTRL+) to place the parens as a pair and cycle through possible beginning locations. Likewise, from the start paren, use the ^( key to type an open paren and toggle between all of the possible matching end positions. (… need to work on this concept, particularly the part about the initial insertion versus toggling positions, maybe with no selection it inserts and makes a selection, and if you don’t want to insert, you must select (or use another key that does both the select and the placement)).

Like insertion, deletion also works differently on the open and close paren/bracket/brace. Deleting the right paren re-opens the paren, whereas deleting the left paren removes both.

## Help, I’m Trapped

While, in general, you can just type unmodified Python, there are two cases where typing as you would in a text editor will trap you in a corner. One is typing on the denominator of a vertically-arranged fraction. LilyPy, by default, arranges fractions in mathematical notation. After typing a division operator (/ or //), your cursor will be in the denominator of the fraction. In a fully-textual language, you “leave” the context of the denominator by typing a lower precedence operator or an end-paren. For a vertically arranged division, that sort of syntax does not work, and you need to explicitly leave the denominator using the tab or arrow key. Likewise, during the entry of block comments, LilyPy saves you from having to prefix every line with #, but at the cost of the Enter key meaning “insert a newline in my comment”, as opposed to “end my comment”. Again, the Tab key is your friend, and you can use it to exit the comment (both from the beginning and from the middle).

## 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 LilyPy, 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 that will create a list when executed. Each time you execute it, it will create a new list, but with a slight difference: a small circle near the top of the bracket. The small circle on a bracket or brace indicates that it represents underlying mutable data. LilyPy allows you to directly edit mutable data in a way that actually changes it. In the above example (typing [2,1]), 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() to the execution result (with the circle on the bracket) and execute that, you would see that the list suddenly rearranges itself from [2,1] to [1,2]. This ability to modify mutable data, allows you to 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 with respect 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, LilyPy tries to ensure that your interactive edits retain the identity of the objects they operate on. Even so, it can be difficult to keep them straight. 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. LilyPy 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 LilyPy’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 LilyPy 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 LilyPy in to a text editor, will also yield standard Python. In the very worst case, should you somehow lose all access to the LilyPy 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 LilyPy 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 LilyPy’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 LilyPy the context to understand what the user is pointing at. Most of LilyPy’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 LilyPy’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, as well as used as a separator character for arguments (such as when a macro needs to take multi-character arguments). 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 lilypy package and call:

lilypy.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), which should take the following arguments:

1. astNode -- The Python AST node that followed and may or may not have been specified via the macro text substitution
2. macroArgs -- The text following the colon as the macro appears in the save file
3. codeArgs – List of ASTs resulting from processing code between macro ($ and $)$ characters (… this may not be available in the user-visible version, as they will be encouraged to create Python code that can stand-alone and just tag it with a macro)
4. window (the window in which the icon should be created).

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). For example, if you want a macro to insert the text “a.b.c”, and to attach its data and icon creation function with the attribute “.b”, the substitution string would be “a$.b.c”.

LillyPy’s use of Python ASTs as an intermediate representation, results in a couple of peculiarities that may affect how you write macros. One is that the icon tree does not correspond perfectly with the Python abstract syntax tree. In particular, LillyPy orders attributes from the root variable and Python from the last call/subscript/attribute, back. Another is that because the macro changes the form of the code (from AST to icon), you can’t apply multiple macros to the same AST. (… there are more peculiarities than this, and more advice is needed for these two. It may even be that user macros should not see ASTs at all, but work instead with icons which can be traversed up and down and go through multiple transformations)

#### Built-in Macros

To help you understand the content of the .pyg file, and avoid conflicting names when users import your .pgi file with ‘\*’. (… I have not fully thought out the mechanism for qualifying macro names, and this may be a minefield, since one character will wholesale change every macro in the file), here are the built-in macros:

**$:**<args>**$** -- “Unnamed” macro. A colon is usually used to separate the macro name from its arguments, and when the macro name is left out, that means it is implied by whatever python type follows it. This is used anywhere the LillyPy environment needs to attach graphical attributes to a Python construct. For example, list layouts can be specified as ‘h’ (horizontal), ‘v’ (vertical), or as a number (number of columns)

**$@$** -- Positions code fragments in the module window. Unlike other macros whose arguments always follow a colon, the first arguments (position) follow directly after the @, as in $@+5+5$. This may be followed by a colon and a single argument of a context type code and a code argument introduced with open paren within the macro and terminated with $)$. For example: “$@+0+0:a($ .attr $)$”. The type code is used to mark fragments that are not valid Python statements or expressions on their own, but may be placed on the window nonetheless to help you compose code graphically. Type codes are:

|  |  |
| --- | --- |
| a | Attribute (including subscripts and function calls) |
| e | Expression. This will not appear in @ but is used in Ctx and Entry macros (where it is the default and can also be omitted). |
| s | “as” clause, which can be (interchangeably) used in “with” and “import” statements |
| d | Dictionary element: a ‘:’ operator for specifying a key/value pair |
| f | Function argument assignment or \*\* |
| c | A clause of a comprehension, such as “for a in range(10))”. (…I think this will eventually only apply to $Entry$ macros, as they will normally be converted to ‘for’ and ‘if’ statements at the top level.) |

**$Empty$** and **$EmptyDict$** -- Stand-ins for empty arguments. This macro simply substitutes a valid identifier (or dictionary element in the $EmptyDict$ case) to allow the Python parser to ignore a missing argument (such as in “[a, ,b]”),and LillyPy to render a highlighted gap.

**$Ctx:<t>($ .. $)$** and **$CtxDict:<t>($ .. $)$** -- Context error. LillyPy allows all sorts of Python errors during code construction. The most common being code that is correct for another context but not for the context in which it is currently sitting. For example: “a, 3 = b, c” can be written in LillyPy (but highlighted as an error), and would be coded in the save file a “a, $Ctx($ 3 $)$ = b, c”. If the context type were not an expression, the type code would also be included. For example, “a:b = c” would be coded as “$Ctx:d($ a:b $)$ = c”.

**$Entry**:<text>**:**<t>**($ $)$**-- Entry/Placeholder icon. <text> is any unprocessed text in the icon and (optional) parenthesis enclose pending arguments with types specified per the table above under $@$. For example, an entry icon containing the word “text” with pending arguments “.x” and “y” would look like: “$Entry:text:a($ .x $)e($ y $”. In the text string, both ‘$’ and ‘:’ are escaped by doubling (‘$$’, ‘::’) (… doubling $ doesn’t work, as we (need to?) allow macros to abut).

### Icon Creation

# Unused

The cases that will orphan more than a single branch are:

1. Removing parens/brackets/braces from a multi-clause list in a context that both surrounds it and prohibits multiple arguments (if, elif, while, subscript and inline-if). For example, deleting the parens around a,b in the code below:

if f(a,b)

will detach both a and b.

1. Adding a limiting statement around an existing top-level comma-separated list. For example, typing “while” before:

a, b, c

will detach a, b, and c because “while” neither accepts a list nor can itself be part of a list

1. Removing parens/brackets/braces from a list that has an attribute attached, when the last element of the list is unable to support attribute attachment. For example:

[1, 2].pop()

will detach 1, 2 and .pop. Note that if there is a reasonable parent to receive the 2, only “.pop()” would then be highlighted for deletion.

1. Editing (putting the text cursor in) the base component of multi-part icon that is not a simple list, such as “for” in “for x in range(10)”. Here, the “in” structure will be preserved until you complete something that can’t use the “in” (anything other than “for” or “async for”) in the text box.