

Product Spec: Lexical Completions

This document describes a language-agnostic best-case implementation for lexical completions.

Background

Lexical completions predict the next tokens of code that a developer will write by training the GPT-2 language model on code. This approach allows us to replicate functionality for an additional language with minimal effort. A downside of this implementation is that it does not understand code structure, so the generated code may be invalid or delimited in a strange manner.

Description

Lexical completions save developers time by predicting what they are most likely to write next. Time is saved by reducing physical keystrokes typed while minimizing the mental capacity needed to parse completions.

Design Principles

These are the foundational principles that guide how lexical completions are designed.

- The most important scenarios are scenarios when users are already waiting on completions
- Lexical completions should not obstruct other completions providers when the other providers can provide user value.
- At any given point, the set of lexical completions returned should maximize expected value added from all providers.
- Completions are as relevant as possible.
- Completions should look as close to what's being inserted into the code as possible to reduce the effort to parse the completion.
- It is clear where the user's selection will be after selecting a completion.

Definitions and Assumptions

Definitions

- **Semantic completions** recommend identifiers that make use of language-specific information to provide only valid completions. These are provided by the most popular IDEs such as Visual Studio, IntelliJ, etc.
- Code generally consists of **syntax tokens**, **identifiers**, **keywords**, **literals** (string or int) and **comments**.
 - Comments are ignored and literals are returned as a snippet placeholder. Thus, completions will return a stream of syntax tokens, keywords, snippet placeholders, and identifiers.
- **Paired syntax tokens** are syntax tokens that must come in pairs for code to be valid: `()`, `[]`, `{}`, `""`, `'`.

Assumptions

- Software engineers find syntax tokens easy to type (muscle memory) but harder to parse than identifiers.
- Software engineers find keywords easy to type (muscle memory).
- Users get the most value when completions can suggest identifiers and literals correctly.
- A completion that has incorrect/missing characters/spaces has no value because it requires a manual edit, which

takes the user out of their flow.

- The vast majority of users and editors auto-close paired syntax tokens. → Open paired syntax tokens inserted by our completions should be closed as well.

Completion Specifications

Basic Specifications

- Must contain at least one identifier or keyword → Completions with only syntax tokens or placeholders are never suggested.
- Must not recommend a close paired syntax token by itself, this includes `)] }`
- Must not recommend an unmatched open paired syntax token
- Must not add characters behind existing characters in user's buffer → If you are in a function call, completions will never add characters outside of the function call
- Must show abstract placeholders for literals clearly like `<number>` and `<string>`, similarly, must not show illegal tokens like `KITE_ILLEGAL`
- Must display a Kite logo or hint to differentiate from completions from other providers
- Must not show probabilities and scores for production users, but these should be shown for internal and beta users.
- Should be correctly formatted (ex. no spaces in unexpected locations)
- Completions with trailing horizontal whitespace (e.g `package`):
 - The trailing horizontal whitespace should be removed from the display text and the insert text
- Completions with trailing vertical whitespace (e.g `main\n\n`)
 - The trailing vertical whitespace should be removed from the display text and the insert text
- Completions with leading horizontal whitespace that matches the buffer (e.g buffer `func handle(w $` with completion `http` note the leading whitespace)
 - Trim the leading horizontal whitespace from the display text and insert text

Display Text

The completion display text consists of the current identifier or keyword, if there is one in scope, and then any additional characters that will be inserted. The precise definition is as follows:

- Identifiers are names for entities, and contain alphanumeric characters and underscores.
- The current identifier is one that the cursor is immediately following. Only one identifier can be current at one time.
- After the current identifier, Any additional characters in the completion that are shown are ones that will be inserted. → Characters that have already been typed will not be displayed in the completion.

Some editors display the display text aligned below where the characters will be inserted because the user's eyes are following their cursor. We should support this behavior where possible.

Situation 1: Cursor is immediately after a valid identifier character

Here, the user is either currently typing out, or just finished typing out, an identifier. Completions should include the current identifier in case the user has not finished typing out their identifier.

```
# Initial state
foo^
```

```
# Completion display text  
fooooo  
foo bar  
foo(bar)  
foo(bar, baz)
```

For the editors that align characters below, the matching prefix of the completion (foo) is directly below the matching characters in the user's buffer.

```
foo$  
fooooo  
foo bar  
foo(bar)  
foo(bar, baz)
```

Situation 2: Cursor is immediately after a syntax token.

Here, the user is going to type more syntax tokens or identifiers. Completions should not include the syntax token since they are all relatively independent of each other.

```
# Initial state  
foo. ^  
  
# Completion display text  
bar  
baz
```

For the editors that align characters below, the completion display text is directly beneath where it would be inserted in the user's buffer.

```
foo.$  
bar  
baz
```

Situation 3: Cursor is immediately before a syntax token.

A user is likely inside a function call or a pair of paired syntax tokens in this case. The behavior is the same as Situation 2, where we don't show the syntax tokens that have already been typed.

```
# Initial state  
foo(^)  
  
# Completion display text  
bar  
bar, baz
```

For the editors that align characters below, the completion display text is directly beneath where it

would be inserted in the user's buffer.

foo(\$)
bar
bar, baz

Low Priority Situation: Cursor is before an identifier

This is low priority because we save few keystrokes due to users replacing only a few characters in an identifier and make it tough to parse as the characters cannot be aligned properly. In this situation, we suffix match to the end of an identifier, but do not go beyond the suffix itself. Here are two examples of a cursor directly before an identifier.

Initial state

^bar

Completion display text

foobar

bazbar

Initial state

bar^foo

Completion display text

barbazfoo

For the editors that align characters below, the completion display text is directly beneath where it would be inserted in the user's buffer.

\$bar
foobar
bazbar

Completions ending in syntax

The model is recommending completions ending in syntax. If the syntax completes a thought, then it's worth showing that syntax to the user. If it does not, we should dedupe it since it makes the completion harder to parse and only saves one keystroke.

Completions ending in whitelisted syntax tokens

Completions ending in this syntax should continue to be shown. In the below example, the model recommends completions that end with a whitelisted syntax token. We show all completions here.

Initial state

f^

Possible Completions

foo

foo++

```
foo--  
  
# Completions shown  
foo  
foo++  
foo--
```

- Note that some deduping will take place in the section [De-duping Duplicate Completions](#)

Whitelist

```
'  
"  
(  
)  
;  
++  
--
```

Syntax not on the Whitelist

In this situation, the model predicted plausible syntax tokens or a placeholder that are an incomplete thought. We remove the completions ending with these non-whitelisted syntax tokens so the user has to parse fewer completions.

```
# Initial state  
f_  
  
# Possible Completions  
foo  
foo.  
foo =  
foo <string>  
  
# Only completion shown  
foo
```

Completions with open ([{

Core takeaway:

** Open parens suggested in completions should be auto-closed. Parens outside of completions are left alone.*

** User never sees the _*

Archived: [previous version](#)

The rules below are *only* for completions with an open ([or { in the completion.

Completions that do not contain the equivalent close paired syntax token are invalid so we don't want to show them.

Also, in nested calls, the user is used to an editor auto-closing their syntax, and does not want to keep track of the syntax pairs. Thus, for some of these cases, we add a closing paired syntax tokens; for some other cases, we hide the completions since they would not be helpful.

No modifications are made to the completions returned by the model, so the display text is the model output.

Situation 1: Completion contains paired syntax tokens (and)

This is the ideal case, as the model has predicted a complete function call.

```
# Initial state
f^

# Completions display text
func()
func(foo)
func(foo, bar)
func(foo())

# Possible final states
func()^
func(foo)^
func(foo, bar)^
func(foo())^
```

- After insertion, the cursor is at the `^` behind the `)` since the model thinks this is a full completion prediction. This is in contrast with the other situations where the `)` is auto-inserted.

Situation 2: Completion contains (and ends with a literal

The model has predicted one argument so far. We show the user that this has been predicted, and close the parentheses for them when it is selected.

```
# Initial state
f^

# Completions display text
func(<string>
func(foo, <number>

# Possible final states
func(<string>) // selection on <string>
func(foo, <number>) // selection on <number>
```

- At insertion, we autoclose the paren, the user's selection is on the literal.
- This completion is useful to the user as it provides the user with a snippet as if it was typed by the user, and after filling in the snippet, the option add another argument, TAB to the end of the completion, or type through a `)`

Situation 3: Completions contains unclosed (and ends with (or alphanumeric character

We close the open parentheses of this function call by adding a `)` after it's selected, and placing the user's cursor before the close parentheses that we just added.

```
# Initial state
f^

# Completions display text
func(
```

```

func(foo
func(foo, bar
func(<string>, foo

# Possible final states
func(^)
func(foo^)
func(foo, bar^)
func(<string>, foo^) // selection on <string>

```

- When inserted, a tab stop is added at the `^`.
 - If there is no literal in the call, the user's cursor is at the `^`.
 - If there is a literal in the call, the user's cursor is on the literal. The tab stop is added such that the user's next TAB goes to that location.
- This completion is useful to the user as it is set up as if the user had typed that argument. The user can then choose to add another, TAB to the end of the completion, or type through a `)`

Situation 2a: Completion falls into Situation 2, but contains many unclosed (

The user is in a nested call, so we just close all the unclosed parentheses.

```

# Initial state
f^

# Completions display text
func(((<string>

# Possible final states
func(((<string>))) // selection on <string>

```

- Since it ends with a literal, we do not add a tab stop.
- When the completion is inserted, the user's selection is on the placeholder.

Situation 3a: Completion falls into Situation 3, but contains many unclosed (

The user is in a nested call, so we will close the outside open parentheses with a close paren, after the user selects the completion, to make it valid.

```

# Initial state
f^

# Completions display text
func(foo(
func(foo(bar
func(foo()
func(foo(), bar

# Possible final states
func(foo(^)^)
func(foo(bar^)^)
func(foo()^)
func(foo(), bar^)^

```

- A tab stop is added for each of the parentheses that we auto-close for the user.
- Tab stops are not added for parentheses that we do not auto-close. Examples 3 and 4 only add one tab stop since we auto-close only one open parentheses.

Hidden Cases: Completion contains a (and ends with any syntax that isn't the)

These cases are not shown because the effort to parse them is more than the benefit gained from us helping the user type out a syntax token. This also leads users to expect behavior where we only add arguments on their behalf, so they never need need to squint to confirm that there is not additional syntax in a completion.

In this situation, the following completions are not shown.

```
func(foo.
func(foo,
func(foo =
```

Type-through of closed paired syntax tokens

Close paired syntax tokens inserted via completions must allow a user to type-through them. This is because users expect this behavior from their editor and other auto-complete providers.

```
# Initial state
f^

# Completion display text
func(

# State after completion selected
func(^)

# State after typing )
func()^
```

Completions containing open ' " `

Completions should not never add unpaired quotes or backticks because editors already pair syntax tokens and unpaired quotes will unexpectedly turn code into strings. Since there is no difference between the open and close quote in editors, we do not do the behavior in [Completions with open \(\[{](#)

Only recommend paired quotes and do not auto-pair quotes if a completion has a single quote.

We only show the valid completions and hide the invalid completions below.

```
# Initial state
text = ^

# Valid completions
'foo'
"foo"
`foo`

# Invalid completions
```



```
'foo
foo'
"foo
foo"
`foo
foo`
```

Multiline completions

Based on the assumption that [A completion that has incorrect/missing... syntax](#) is of no use to the user, we want to insert the code with the right formatting for the user.

The display text shows the newline character (↵) for two reasons:

- Users know their completion will update formatting. Without the character, it's confusing whether a space is a space or a newline
- This is a more language-agnostic implementation. In some languages, such as Python, a newline means the end of a statement, so we want to make sure it is clearly communicated to the user.

The completion display text shows the newline character. On insertion, the correct newlines and indentation is applied, and the newline character is not shown.

```
# Initial state
const todo = (state = [], action) => ^

# Display text
{ ↵ switch(action) ↵ }

# Inserted Final state
const todo = (state = [], action) => {
  switch(action)
}^
```

Display Text

There should be a space before and after each newline to make it readable for the user. The two scenarios below illustrate cases with spacing.

```
# Initial state
r^

# Display Text
render( ↵ <Provider # Space before and after the ↵
render(↵<Provider # Incorrect, no spaces before and after

# Inserted Final State
render(
  <Provider
```

```
# Initial state
foo: b^

# Display Text
```

```

bar, ← foobar: barbaz # Correct, space between , and f
bar,←foobar: barbaz   # Incorrect, no space between separate lines
bar,  ← foobar: barbaz # Incorrect, two spaces makes user wonder if bug

# Inserted Final State
foo: bar,
foobar: barbaz

```

Indentation

Indentation, and associated spaces, are not shown in the display text.

```

# Initial state
i^

# Display Text
if (foo) { ← bar   # Correct
if (foo) { ←   bar # User already knows newline;
                # extra spaces feel like a bug, also wastes space in UI

# Inserted Final State
if (foo) {
  bar^
}

```

Line break considerations

Do not show automatically trigger completions here if the entire completion goes on the next line. It looks unnatural to the user, who just wants to type a newline.

```

# Initial state
this.editor = null^
this.editor = null ^

# Display text
← this.editorId = uuid() # Invalid, entirety of completion goes on next line

```

Maximum number of newlines in a row

There should be a maximum of one newline character in a row. Here are some examples that work and some that do not:

```

# Initial state
foo^

# Completions that meet criteria
foo ←
foo ← bar ←

# Completions that do not meet criteria
foo ← ←
foo ← bar ← ←

```

Must continue adhering to existing rules

This behavior must not break existing rules, many of which are outlined above and below this section. Only editors that pass all the spec rules will support multiline completions. Only when all spec behavior is confirmed to work in each editor will this feature be turned on.

I have outlined some common cases below, but this is not an exhaustive list.

- Any [Completions with open \(\[{](#) must continue to auto-close this syntax. Here, I show acceptable and ideal states, as well as a “not acceptable” state.

- ```
Initial state
if (la_

Display text
language) { ← dispatch

Inserted Final state
if (language) { # Acceptable
 dispatch_}

if (language) { # Ideal
 dispatch_
}

if (language) { # Not acceptable, as } is not auto-closed
 dispatch_
```

- Note: All type-through behavior needs to work, as outlined in [Type-through of closed paired syntax...](#)

- [Must not add characters behind...](#) existing characters in user's buffer.

- ```
# Initial state
dispatch (f_)

# Possible Completions
foo      # Allowed
foo) ← } # Not allowed if the ) is the same as in the buffer
          # and the } is after characters in the user's buffer
```

- [Completions ending in syntax](#) only show if the ending syntax is on the whitelist.

- ```
Initial state
f_

Possible Completions
foo { ← bar. # Allowed
foo { ← bar, # Not allowed, since , is not a whitelisted syntax token
```

- [De-duping Duplicate Completions](#) works for completions that end with `[ ] { }`

- ```
# Initial state
this.s_

# Possible Completions
setState({ ← foo: bar
setState({ ← foo: bar ← }
```

```
# Deduped Display Text  
setState({ ← foo: bar ← })
```

- [🔗 Must not recommend a close...](#) syntax token by itself.

```
# Initial state  
b^  
  
# Possible Completions  
bar      # Allowed  
bar ← }  # Not allowed, as it has a closing brace } by itself
```

Fallback behavior

If the completion does not fit into a formatting rule, we insert the completion as is.

Placeholders

Placeholders look different between lexical-only languages and multi-provider languages because it's important that all placeholders for a language look similar.

- In lexical-only languages, we use [🔗 Blank Placeholders: 📄 Product Spec: Completions Mixing](#)
- In multi-provider languages, we align the design to the [🔗 Static Placeholders: 📄 Product Spec: Completions Mixing](#) that semantic providers use as they provide more user value than tab stop placeholders.

For in-depth notes on the design below, see [📄 Placeholders Proposal](#)

Lexical-only languages

We use [🔗 Blank Placeholders: 📄 Product Spec: Completions Mixing](#), which are represented by the ellipsis icon `...`, as the placeholder for string and number literals. Note that this is different from three periods in a row. Tab stop placeholders do not insert any text in the buffer when selected. The completions UI is not triggered since the lexical model does not have any completions to provide.

Note: [Vim does not support the ellipsis icon](#) `...` so we instead use three dots in a row `...`

Example 1

Here, a completion contains a blank placeholder between single quotes. After a user selects the completion, an empty string is inserted. The cursor is on the tab stop between the quotes.

```
# Initial state  
import foo from ^  
  
# Completion UI  
'...'  
  
# After selection, final state  
import foo from '^'
```

- The user's cursor is between the quotes
- The completions UI is not triggered since lexical does not have any completions to provide.

Example 2

Here, the completion contains a placeholder, then some syntax and identifiers. After a user selects the completion, an empty tab stop is inserted instead of the placeholder. The cursor is on the tab stop. A user can enter some text, and then TAB to the end of the completion.

```
# Initial state
foo: ^

# Possible Completions
..., bar

# After selection, final state
foo: ^, bar

# User types a number
foo: 5^, bar

# User presses TAB
foo: 5, bar^
```

- The user's cursor is inserted at the placeholder. After typing, the user can then TAB to the end of the completion.
- The completions UI is not triggered since lexical does not have any completions to provide.

Multi-provider languages

Lexical placeholders match the design and behavior of those from the semantic provider, and so are represented as [Static Placeholders](#): [Product Spec: Completions Mixing](#).

For Python behavior, see [Static Placeholders](#): [Product Spec: Lexical Python](#)

Example 1

Here, the user sees completions for the function `foo`. The lexical provider provides a static placeholder as well as a completion that goes onto the next line. To the user, both placeholders look and function in the same manner.

```
# Initial state
f^

# Possible Completions
foo(<int>): ↵ bar      # Lexical provider
foo(<my_x>)      # Semantic GGNN provider

# After selection of the first completion, final state
foo([int]):          # Selection is on [int], completions UI shown
  bar

# User types my_y
foo (my_y^):
  bar

# User TABs to end of completion
foo (my_y):
```

```
bar^
```

Language-specific filters

Javascript

If any completion has a `>`, it has to end with `</ident>` or `/>`. Otherwise don't return it.

Here is an example of what we filter out, where the first is not shown but the second completion is.

```
# Potential completions
<h1 className=""> # Not shown
<h1 className="" # Shown
```

It is ok to show the second completion because once the user types `>`, most editors will auto close the jsx element.

Completion Triggers and Situations

Compared to [Triggers](#): [Product Spec: Editor Plugins](#), in lexical, completions trigger for every keystroke.

Generic

- The completions engine should not return completions when the user is inside a comment.
- The completions engine should not return completions if it is likely that the user wants to insert a newline
 - This is based off the character immediately to the left of the cursor, there are “suppression after” character lists for each language (see below).

Javascript

Single char blacklist

We do not suggest completions where it's probable that the user will insert a new line.

If the following characters are immediately to the left of the cursor, that we do not show completions for. This is approximated by looking at Javascript, React, and Vue files and identifying the non-alphanumeric characters that are at the end of lines.

```
(){}:;,+->\n
```

Here is an example of how this works in practice. In the below example, we have completions since it's a common case for a user to write the expression on multiple lines.

```
# Initial state
const post = () => ^

# Possible Completions
{ ← switch (action
{ ← return action
```

After the user types the blacklisted character `{` no completions show. It is possible that this function can be expressed on one line, but since there is a likelihood that the user may want to insert a newline, we block completions in these scenarios.

```
# Initial state
const post = () => {} # Closing brace is usually added automatically

# Filtered out Completions
switch ( # Do not show these
```

Line situation

React and Vue code is often between quotes or backticks. Thus, completions are shown when the user has their cursor between the following:

```
' '
```

- Triggering on the insertion of quotes could lead to completions showing after someone types a closing quote.

For example, a user who types single quotes will not see completions yet, but will see completions only after typing a

```
' ' # no completions yet
'a' # completions show
```

Comments

No completions are shown when a user is writing a comment. In the below scenarios, no completions are shown.

```
// foo_
```

```
/*
foo_
*/
```

Python

[Automatic Triggers](#): [Product Spec: Lexical Python](#)

[WIP] Golang

Situation with text on same line

Low Priority situation: Cursor has a valid identifier character immediately after

The user is likely in the middle of deleting characters or correcting the spelling of an identifier. Completions are not valuable here, so we do not show any completions, but if we previously did a suffix match, then it's low priority to hide those completions. These are examples of this case:

```
_foo
```

```
foo^bar
```

All other situations:

The user is actively typing their line of code, so we want to show to show completions. If there are in the middle of a line of code, it is important that we [Must not add characters behind...](#) existing characters in user's buffers. Here are scenarios that we should show completions:

```
^foo^
foo^
foo(^
foo(^, bar)
foo.^
^foo
<string>^
```

Rules to show less or more tokens in a completion based off of surrounding characters (eg. in an empty function call vs. in the middle of a line) are not included here, because the additional complexity in implementation and overhead is not worth the benefits at this time.

Completions List Specifications

Show two lexical completions maximum

Lexical Completions Ranking

Lexical completions are ordered by expected value added. Expected value is defined as $\text{char inserted for the user} \times \text{likelihood that completion is correct}$

De-duping Duplicate Completions

Two completions are said to be duplicates if they differ only by lexical (non-valid-identifier-characters) characters at the end of the completion. Here is an example of two “duplicate” completions:

```
foo(bar
foo(bar)
```

The below is not a duplicate completion because `x` is a new identifier.

```
foo(x
foo()
```

In these situations, we only show the longest completion since the user finds the most value in the longest completion.

Note: The completions being evaluated here end in syntax that exists in the whitelist listed in [Whitelist](#)

Here is an example where the model predicts the function or array or entire function, but also returns the function or array name by itself. The user finds the longest completion, the full empty function call, the most valuable here.


```

# Initial state
f^

# Possible Completions
func
func(
func()

# Deduped Display Text
func()

# Final state
func()^

```

Here is an example where the last `)` is not present. The user finds the longest completion, the one with the function call parentheses, more valuable than the one without.

```

# Initial state
f^

# Possible Completions
func
func(

# Deduped Display Text
func(

# Final state
func(^)

```

Here is an example with an argument in the function call. In this example, if the model predicts that this is a complete function call, the user finds that more value so we only show that one.

```

# Initial state
f^

# Possible Completions
func(foo
func(foo)

# Deduped Display Text
func(foo)

# Final state
func(foo)^

```

Low Priority example: We only show the longest completion in this placeholder example.

- It's not a high priority to make sure the below adheres to the spec.

```

# Initial state
f^

```

```
# Possible Completions
func(<string>
func()

# Deduped completion
func(<string>
```

In the future, we will identify syntax combinations that we may want to keep. For example, the following are completions that we want to show, but they are not a priority right now.

```
foo++
foo--
```

Completion UI

Completion Icon

The icon is on the left of a completion. For editors that allow custom icons, we want the Kite icon to match the other completions.

Editors that do not have icons or do not allow custom icons

We maintain the existing behavior.

VS Code keeps showing the existing icon Sublime and Vim continue not showing any icon.



Editors that allow custom icons (Lexical-only languages)

Lexical-only languages such as Javascript and Golang show a centered Kite icon, in a size that matches other icons, and on the default color, or neutral color, background. The goal is to have the Kite icon look like it belong with the icons of other completions.

Atom

Kite logo is centered.

Same Kite logo size as before since there is ample padding to the sides.

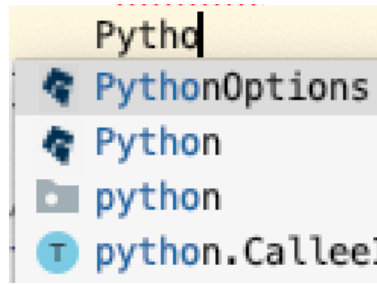
Background color is dark grey.

Jetbrains

Kite logo is centered.

Kite logo fills to fit the space afforded to the icon so it's the same size as other icons.

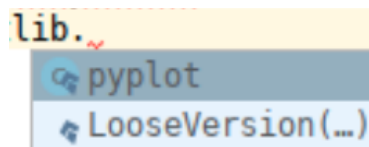
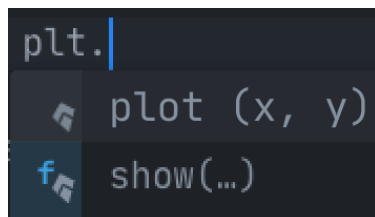
Background color matches the editor default.



Editors that allow custom icons (Mixed Lexical and Semantic languages)

For mixed languages like Python (Semantic + Lexical), the lexical completions maintain the same Kite icon size, positioning, and background as semantic completion counterparts so completions from Kite look uniform. However, there is nothing “behind” the Kite icon. This behavior is similar to what was implemented in [PR #610](#)

Atom	IntelliJ
Icon positioning remains the same.	Icon positioning remains the same.
Icon size remains the same.	Icon size remains the same.
Background color is neutral (not type-specific).	Background color is the default editor background.
Nothing “behind” Kite icon for lexical completion.	Nothing “behind” Kite icon for lexical completion.
In this example, the top completion is a lexical completion while the second is a semantic completion.	In this example, the top completion is a semantic completion while the second is a lexical completion.



Completion Hint

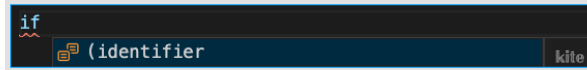
The hint is on the right side of a completion. It can consist of type information, probability scores, and Kite branding. We do not show the type information or probability scores for lexical completions, but maintain Kite branding.

Type information

No type information is shown as lexical does not know the completion type. Here, you can see the comparison:

Lexical

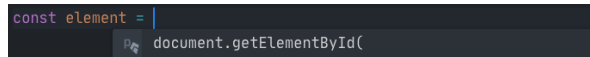
Semantic



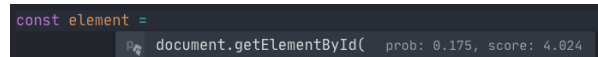
Probability Scores

No scores are shown for the user since users do not use the score to decide on whether or not to select a completion. It just adds visual noise.

No scores (Launch behavior)



With scores (Internal/troubleshooting behavior)

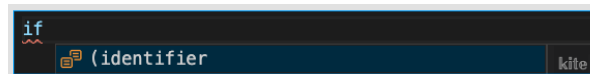


Kite Branding

We want users to know which completions are from Kite. This branding in the hint is because the editor does not allow Kite branding in the icon. Thus, we maintain the Kite branding for editors that have it.

VS Code

Continue showing the stylized *kite*



Sublime and Vim

Continue showing Kite icon ◇ (can't find the exact icon)

- Code below is not Javascript since editor support was not ready at time of writing spec.
- We do not intend to launch with Vim support. When we do, we can implement this.



Performance Considerations

- Must return the complete list of completions in 100ms or less. 100ms is the threshold where interactions feel instantaneous ([mentioned in this startup's marketing material](#)).
- Must show show completions from other providers if lexical completions are not available at the 100ms cutoff. → Continue generating completions and these will likely show after the user's next keystroke
- Must not impact typing latency

Metrics

Holistic Completions Measurement

Our goal is to maximize completions selected by all providers, so we should measure the total completions shown, and

selected, from all providers.

Implement `selected_other_prov` and `at_least_one_shown_other_prov`

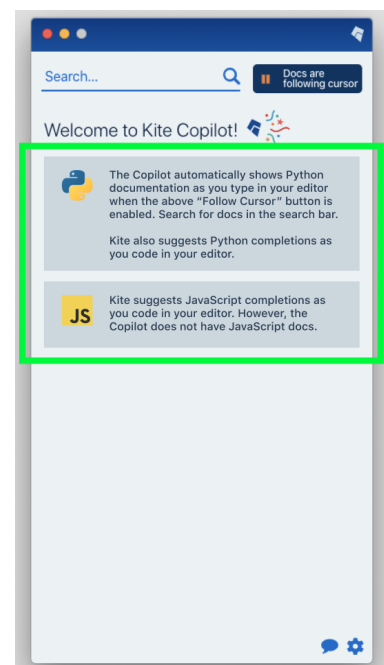
- This metric is gathered only when Kite completions are returned.
- Only one editor is needed as this is likely technically difficult and used to see if our changes are actually making the user's experience better
- A new metric was chosen instead of adding another provider source so all existing charts are still solely about Kite completions.
- A new dashboard for holistic selected/shown rate would be calculated by:
`(selected + selected_other_prov) / ('at_least_one_shown' + at_least_one_shown_other_prov)`

[Production] Non-completions UI

Dashboard

We update the dashboard to be specific about what functionality is specific to Python, but also explain that users get completions from lexical languages. Every user sees the same content, regardless of the language they use.

The sections are broken up into sections that describe Python behavior as well as Javascript behavior.



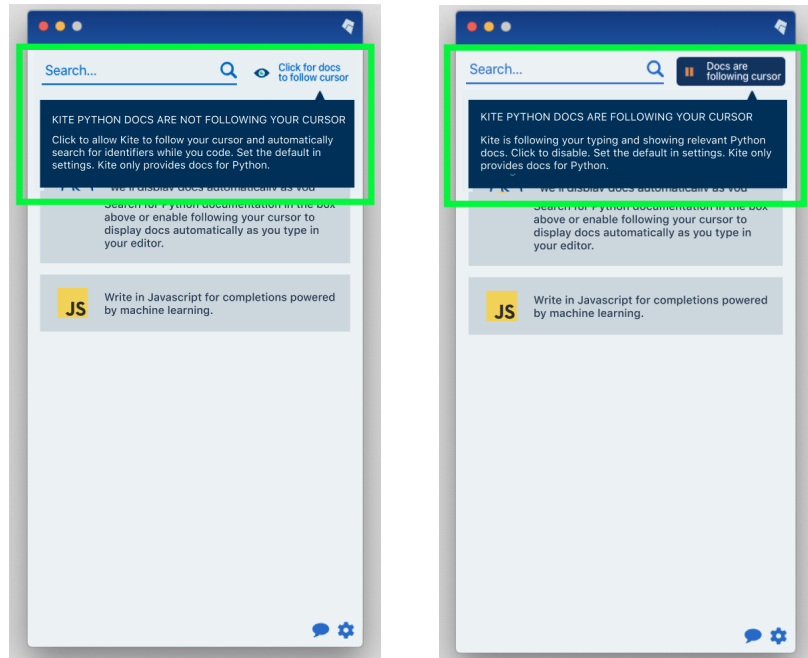
Figma design

Follow Cursor

The tooltip that shows on hover of the Follow Cursor button now mentions Python explicitly in the title and content.

The text weight is "Regular". The

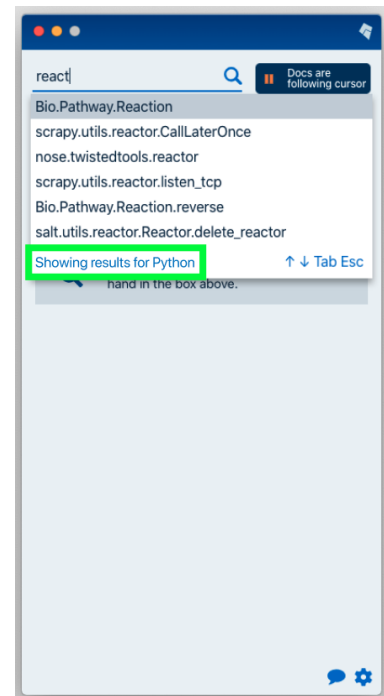
previous text had light weight and was hard to read.



Figma Design

Search Bar

In the bottom left, a status message tells the user that the results are from Python.

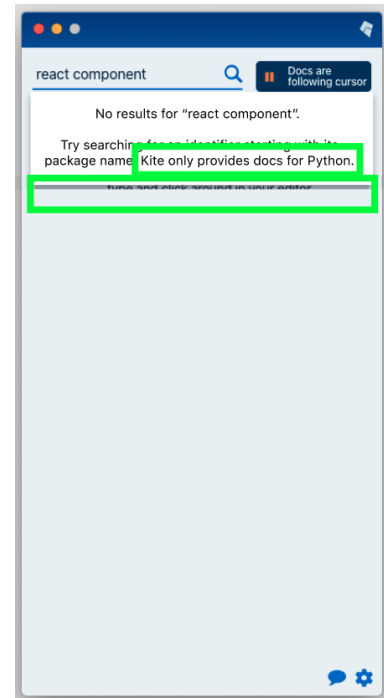


Figma Design

For searches that have no results, additional text tells the user that docs are

for Python only.

- The bottom row is removed. Users do not need to know about the keys used to select search results.
- The text now has spacing between the first and second sections.
 - The spacing allows us to separately communicate the result of the search from the troubleshooting notes. Thus, users who hit this message again only need to read the first section.



Support Flow

Once this is ready, it can go out to users.

The chat bubble icon now opens the [Feedback and Help Page](#).



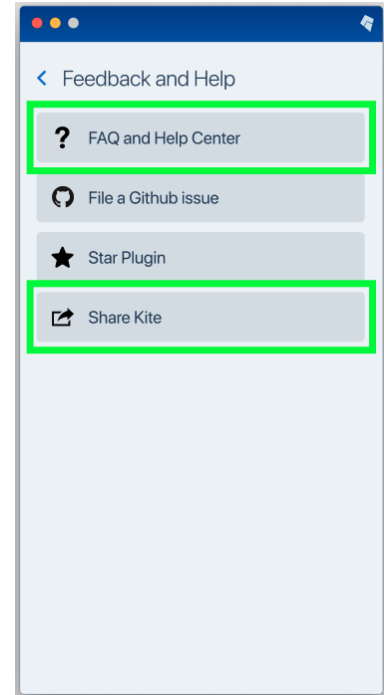
Feedback and Help Page

This page implements some of [Product Spec: Support v2](#).

All rounded rectangle sections here are clickable components. Hovering over them has a color change as described in [Hover State](#)

FAQ and Help Center opens <https://help.kite.com/>, and in the future, will point to [Self-Help Flow](#): [Product Spec: Support v2](#)

Share Kite opens <https://kite.com/invite/>

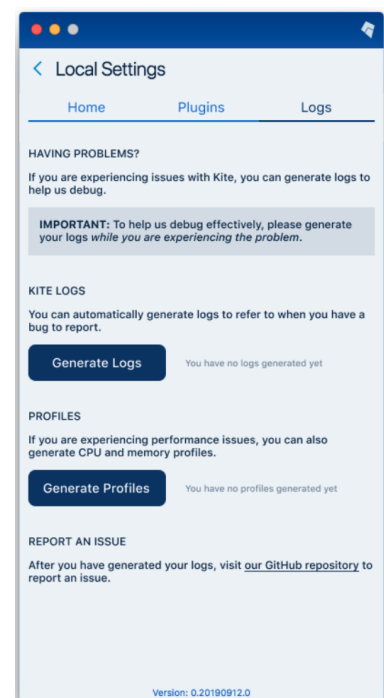
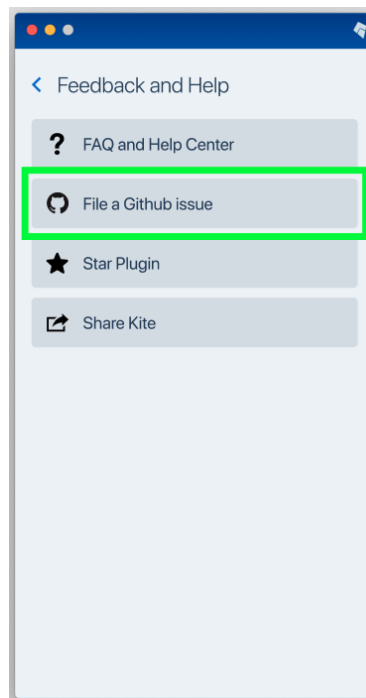


Figma Design

File a Github issue

1. Opens <https://github.com/kiteco/issue-tracker>
2. Brings the copilot to focus with a 1000ms delay so this is brought into focus after the browser page opens. It also opens the **Logs page**.

In the future, this will point [Copilot Flow](#): [Product Spec: Support v2](#)



Nice to have:

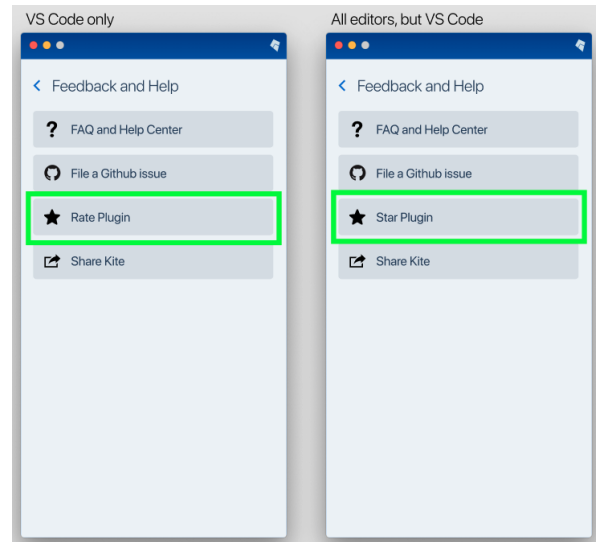
The third section shows a different title based off of most recently used editor. This is because VS Code is the only editor that has a marketplace where users can rate the Kite plugin. Also, because engineering already

implemented this.

Star Plugin opens a link to that editor plugin's Github location, or the Kite plugins page if no editor page.

- Atom: <https://github.com/kiteco/atom-plugin>
- Sublime: <https://github.com/kiteco/KiteSublime>
- Vim: <https://github.com/kiteco/vim-plugin>
- IntelliJ, Spyder: <https://github.com/kiteco/plugins>

Rate Plugin opens Kite's page in VS Code's marketplace at <https://marketplace.visualstudio.com/items?itemName=kiteco.kite&ssr=false#review-details>



Navigation

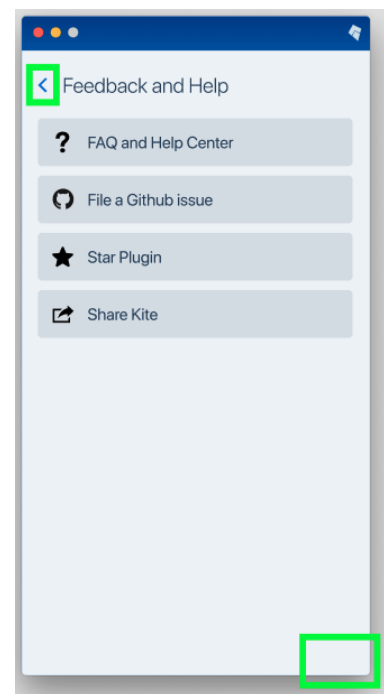
The button in the top left goes to the [Dashboard](#)

The chat bubble is not shown in the bottom right corner since this Feedback and Help page is already open.

Nice to have:

The settings icon is not shown in the bottom right corner. This means all “back buttons” go back to the dashboard rather than the most recent page. Users know where each button click will go.

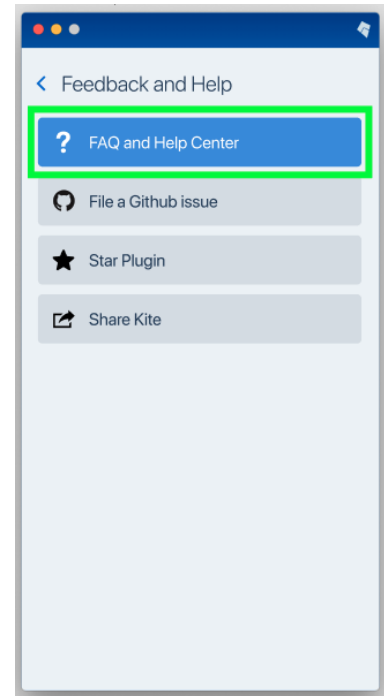
It's also important as we add additional pages like coding stats so that we don't need to remember the user's most recently visited page.



Hover State

We want the user to know that these are buttons. The hover state uses new colors that haven't been used in the copilot since the existing orange color is too strong for such large buttons.

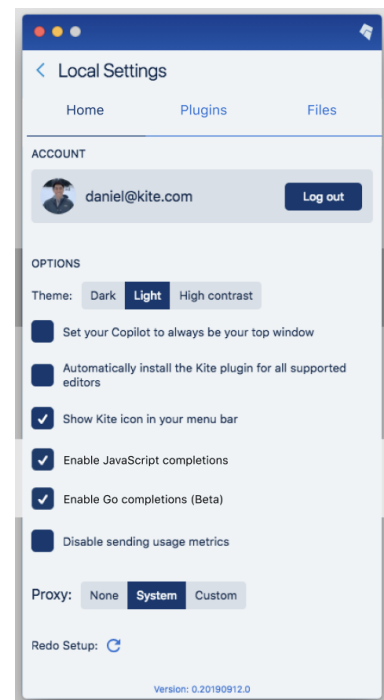
See [Color usage in copilot](#)



Settings

The languages can be disabled in the settings pane. When a setting is disabled, no feedback dialog is shown.

When a language is launched, the text “Beta” is removed from its respective setting. In this example, Javascript is launched but Go remains in beta.



Menubar and Taskbar icon

When this is ready, this can go out to production users.

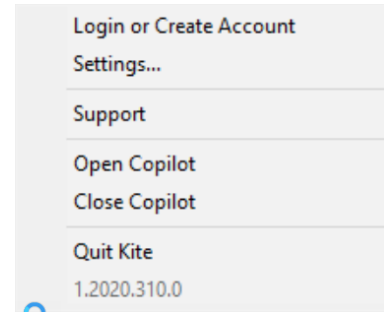
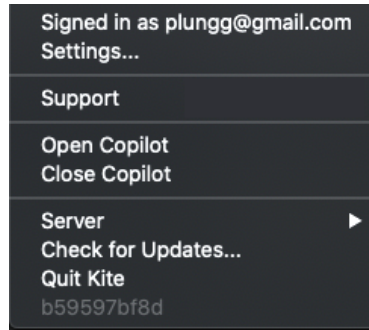
There is no longer additional a “side dropdown” at **Support** for Mac and Windows users. There is no accessible location

for Linux users so they are not affected.

In Lexical, **Support** now opens the Copilot [Dashboard](#)

In Python, **Support** now opens the Copilot [Python Help and Feedback Page](#).

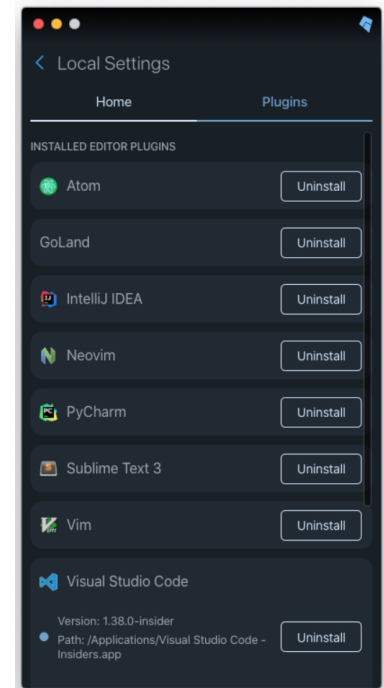
Since this change affects Python as well, this change should only be made if [the Python Feedback and Help page](#) is done.



Plugin Manager

All supported editors are listed here. There is no distinction between editors that support Python or another language, as what matters is your file extension.

The only editors that are grouped together are if they are the same editor, but can run independently of each other (eg. Beta build of another editor). Thus, JetBrains IDEs are not grouped together.



Plugins Language Support

All editors support all languages supported by Kite to the best of their ability.

- Atom, Sublime, Vim, and VS Code will support all languages.
- Spyder will only support Python.
- JetBrains editors' language support is up to engineering.
 - IntelliJ IDEA supports Python and Javascript.

For example, IntelliJ will support Kite for Python, Javascript, and Go. The behavior for languages unsupported by Kite, say Java, is the same as before. For IntelliJ, that means no icon or status will be shown.

The onboarding experience for languages can be found in the following sections:

- [🔗 Onboarding Experience \(Beta\)](#)
- [🔗 Previously unsupported editors \(Beta\)](#)
- [🔗 Production Onboarding Experience](#)

Kite status

The Kite status icon will display these messages, as appropriate per editor. This should be in black font color, rather than red, since they don't require the user to make changes.

Message	Condition
Kite: Disabled	If a language supported by Kite is disabled by the user

Kite should show, and obey, existing status messages when possible:

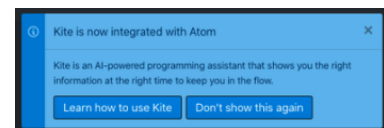
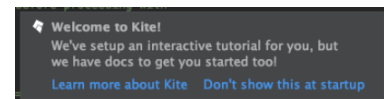
Message	Condition
Kite: Not installed	Kite Engine is not installed.
Kite: Not running	Kite Engine is not running.

The exact wording or capital casing is dependent on the editor.

[Production] Welcome Notification

There is no change to the current experience for Atom, IntelliJ, PyCharm, Spyder, Sublime, Vim, VS Code since their welcome notifications are language-agnostic, or do not show.

- Atom, IntelliJ, VS Code, Spyder, and Vim editors continue to show the “Welcome to Kite” notification on start the first time a user opens the editor that Kite is installed.
- Sublime does not have a “Welcome to Kite” flow because it only has blocking dialogs, which are very strong.



Example welcome notifications

Goland and Webstorm

Current behavior: Goland and Webstorm only open the language beta notification.

Proposed behavior: Show the language-agnostic welcome notification.

- **Title:** Welcome to Kite
- **Content:** We've setup an interactive tutorial for you, but we have docs to get you started too!
- ([source](#), [IntelliJ's welcome notif](#))

[Production] Tutorial File

- Some editors are cross-language. For these editors, we show the Python tutorial on start. Inside the tutorial, we show the user choose how to easily switch to a tutorial for a different language.
- Other editors are language-specific. For these editors, we show the language-specific tutorial file immediately. It's a nice-to-have to not show options for other languages.

Cross-language Editors (Atom, Sublime, Vim, VS Code)

Vim is mentioned here for completeness although it won't be supported at JS launch

We show the Python tutorial by default and tell users how to access other tutorials easily. This is done by creating commands to access language-specific tutorials as well as adding a section to the tutorial file itself.

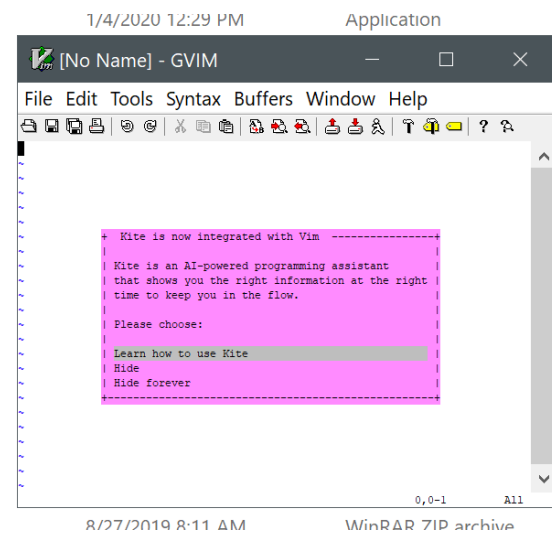
Showing the Tutorial

This section is about showing the Python tutorial the first time a user opens the editor with the Kite plugin installed.

No changes are needed for Atom and VS Code as they already do this.

- VS Code has an issue with the tutorial not opening [#10753](#)

No changes are needed for Vim, which shows the opening notification, then allows a user to select the tutorial. This fits the workflow described in the spec.



Selecting “Learn how to use Kite” opens the Python tutorial

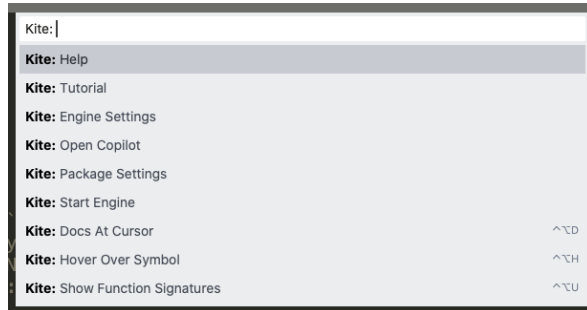


Changes are needed for Sublime,

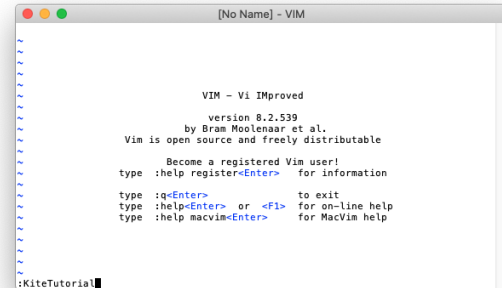
- Current (recently implemented) behavior is to only show the tutorial file when a user edits in a supported language
- Proposed behavior is to revert this change and show the Python tutorial on start

Tutorial Commands

This section is about adding or updating Kite commands for tutorial files with different languages. Here are examples of the command palette and command line.



Sublime's command palette



Vim's command line

Current behavior:

- Atom, Sublime, and Vim have a command/action for opening the tutorial
- VS Code does not have a command for opening Kite's tutorial, though it has commands for other Kite functionality.

Proposed behavior:

- Atom and Sublime:
 - Rename the command/action that opens the Python Tutorial as shown in the table below
 - Add commands/actions for Go and Javascript
- VS Code:
 - Implement ability to open tutorial from command palette
 - Add commands for the Go, Javascript, and Python tutorials
- Vim: No changes for now as Vim as Vim does not support Kite for Go or Javascript.

	Command for Atom, Sublime, VS Code	Command for Vim (in the future when we have Go/JS support)
Python	Kite: Python Tutorial	KitePythonTutorial
Go	Kite: Go Tutorial	KiteGoTutorial
Javascript	Kite: Javascript Tutorial	KiteJavascriptTutorial

Changes to Python Tutorial File

A new section informs the user what language this tutorial is in as well as how to change to a different language tutorial.

JetBrains family of editors (IntelliJ, Pycharm, Goland, Webstorm)

We show the tutorial with the right language by default and tell users how to access other tutorials easily. This is done

by creating commands to access language-specific tutorials as well as adding a section to the tutorial file itself.

Showing the Tutorial

No changes need to be made for Goland, IntelliJ, and PyCharm as they open the tutorial to their respective languages (Go, Python, and Python respectively).

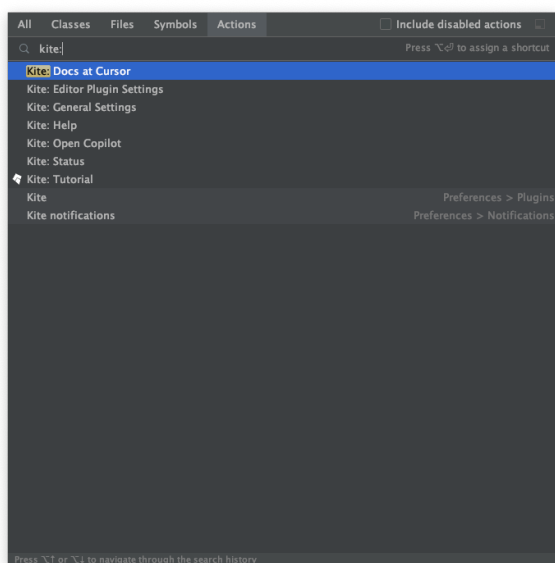
@Patrick Lung to spec out single language case, as well as multilanguage priority order

For Webstorm:

- Current behavior: Does not show a tutorial file.
- Proposed behavior: Show a Javascript tutorial file.

Tutorial Commands

This section is about adding or updating Kite actions for tutorial files with different languages.



IntelliJ's actions

	Action Name
Python	Kite: Python Tutorial
Go	Kite: Go Tutorial
Javascript	Kite: Javascript Tutorial

Note: Kite logo for these tutorials, make GH issue to add Kite logo to other actions

Changes to Python Tutorial File

A new section informs the user what language this tutorial is in as well as how to change to a different language tutorial.

Language-specific editors (Spyder)

These editors should not show tutorial files for different languages.

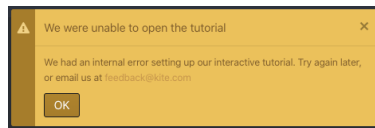
No changes are necessary for Spyder since it opens the Python tutorial by default.

Unable to open tutorial file

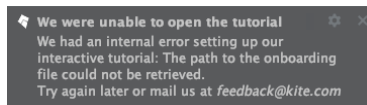
If we are unable to open the tutorial file, for the editors that have it, we show the user some notification that the tutorial could not be opened. This could occur if a user uses a command to open Kite's tutorial while kited is not running.

No change is needed for Atom, Vim, or JetBrains (Pycharm, IntelliJ, Goland, Webstorm).

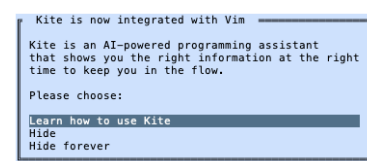
Atom



Pycharm



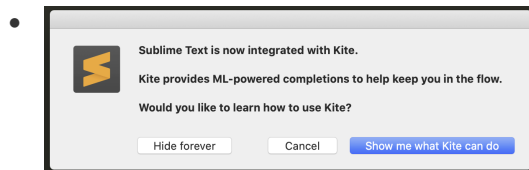
Vim



This opens the Vim Help section.

Nice-to-have: This isn't high priority since it's a rare case that a user encounters this.

- Sublime: Remove “for Python” from this message, so that it looks like:



- This is different from the Atom and Pycharm messages because Sublime does not have a “Welcome to Kite” notification. Thus, it's possible that this will show on the user's first time opening Sublime, and we care for their onboarding experience to look as seamless as possible.
- VS Code: Add the below error message
- Atom, Pycharm: Update the notification text to the below.

Title: We were unable to open the tutorial

Beta Changes

We will enable Kite for [language] beta by default for our existing users, or a subset of them, and provide them with the mechanism to disable the beta. **Below is an example for JavaScript.**

Onboarding Experience (Beta)

Users who are working in JavaScript in a previously-supported editor will see the following experience. A “Welcome to the beta” notification is shown, but a tutorial file is not since the expectation is that the user has been using Kite already in the editor.

Note: See this section for [Previously unsupported editors \(Beta\)](#)

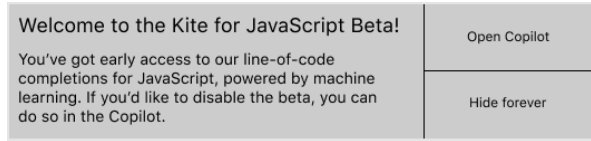
Editor welcome Notification (Beta)

[JavaScript Notification design](#) is on Figma.

This notification is shown the first time a `.js`, `.jsx`, and `.vue` file is opened, per editor and per session. Thus, it is **not shown** if a user:

- Opens a second JavaScript file in the same editor
- Opens a second editor window with a JavaScript file
- Restarts the editor with only Python files open

This behavior is specific per language. A user who has clicked “Hide Forever” for JavaScript will see the beta notification for Golang.



Notification will be updated to fit the styling of each editor.

Notes for [Previously unsupported editors \(Beta\)](#)

Copilot Dashboard (Beta)

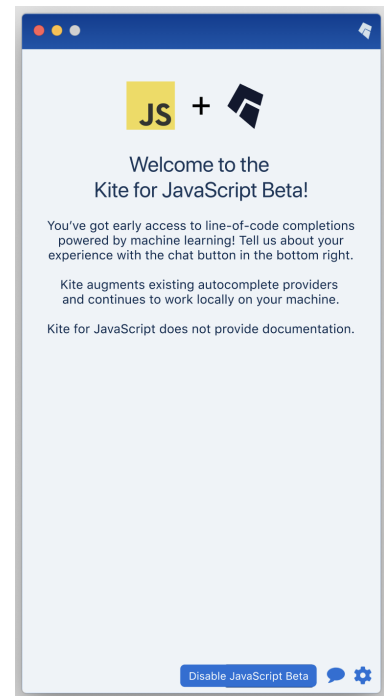
Previously, we had a beta settings checkbox. We removed it since the only users were just disabling settings in setup. Actual users will see this copilot dashboard screen. You can see the past spec description at [Old Beta Settings Checkbox](#)

[JavaScript Copilot design](#) is on Figma.

Shown for all users who are in a `.js`, `.jsx`, and `.vue` file and have the beta setting on.

“Disable JavaScript Beta” button

Clicking this button in the bottom right starts the [Disable Language Experience \(Beta\)](#), but does not affect any other language.

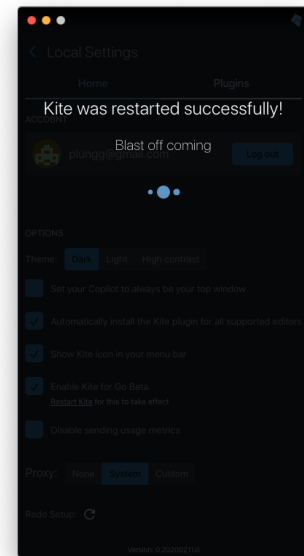


Disable Language Experience (Beta)

If a user disables the JavaScript beta, whether by the [Settings Checkbox](#) or the [“Disable JavaScript Beta” button](#), we first disable the beta and restart the copilot so those changes are taken into effect.

If the user has written some JavaScript, then we ask for feedback to

understand why they are writing in a supported language but disabled completions.



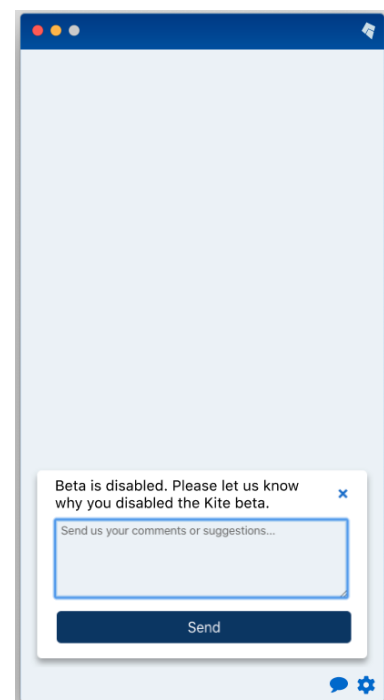
This dialog for feedback is shown after the beta is disabled as this is respectful behavior similar to how email unsubscribes behave.

After the copilot restarts, it opens up to the experience on the right. The copilot screen text is hidden so only the feedback dialog is shown. This way, the user is not overwhelmed by too much text. The user has these actions:

- Click "Send"
 - We show a status message "Feedback submitted" to let the user know that the feedback was submitted successfully. To close the dialog, they can do either of the actions below.
- Close the dialog with the "x"
- Close the dialog by clicking anywhere else in the copilot

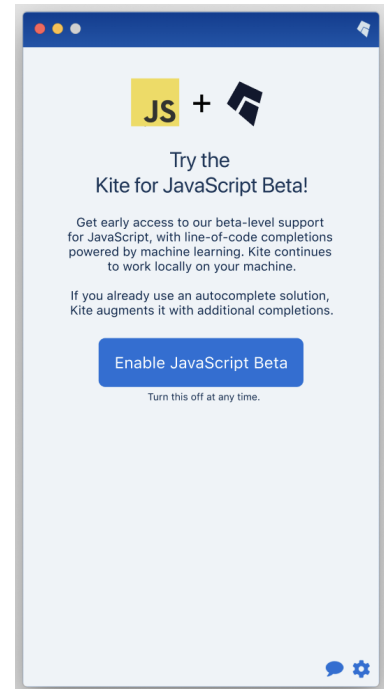
About the dialog:

- It does not close if the user starts editing in an editor even in another language
- It does not persist. Thus, the user no longer sees the dialog regardless of if they submitted feedback or if the copilot is closed.
- Feedback submitted from this dialog is tagged with the last language that a user was using in the format: `[language family extension]-beta-disabled`. Go would be tagged `go-beta-disabled`, while Javascript, Vue, and JSX are all tagged `js-beta-disabled`



After the dialog is closed, the copilot shows the right language screen as described in [Copilot showing the right screen](#).

For users who disabled the JavaScript beta, they most recently were in a `.js`, `.jsx`, and `.vue` file. The screen to the right is the one that they will see.



Telemetry (Beta)

Users who disable a beta will have that setting sent in `kite_status`. We care about three states:

- Beta turned on by Kite
- Beta turned on by Kite, turned off by user
- Beta was not turned on by Kite

Partial Rollout (Beta)

When we do a partial rollout, all of the above behavior is hidden for users not in the rollout. Thus, they will not see a checkbox or screen to enable the beta. This is because the feature isn't ready for everybody, and we only need a limited set of feedback. We only have one opportunity to make a first impression, so want to save that until we're ready.

Previously unsupported editors (Beta)

This behavior is only for previously unsupported editors when a language is in the beta phase.

Primarily, this applies to JetBrains's language-specific editors such as GoLand and Webstorm. Guidance will be changed for cross-language editors where it's not clear what language the user is likely to use.

Onboarding opens the `.js` tutorial file since a user has not used Kite with that editor before. This triggers the "Welcome to JavaScript Beta" notification. Since we don't want to spam the user with two notifications, the generic "Welcome to Kite" notification is not shown.

In the next session, the tutorial file will no longer show. However, if a user opens a `.js` file, the "Welcome to JavaScript Beta" notification will show up as long as they haven't clicked "Hide forever".

Please see [PR #646](#) for more info on behavior.

Design notes

Color usage in copilot

Principles:

- Reuse current product colors as much as possible.
- Only new scenarios will start using brand colors.

Dashboard Hover

On hover, we use a different background color since orange is too strong for a large clickable area with text and icons. Brand colors from [Kite Brand Guide \(Dec 2019 version\)](#) are used in this situation, but text and icons remain black/white.

- Light
 - Text and Icons = White
 - Background = #4A90E2
- Dark
 - Text and Icons = Black
 - Background = #D3D3D3
- High Contrast
 - Text and Icons = White
 - Background = Black

Future functionality

[WIP] Scoring short completions

This section represents the current state of the product. After recent feedback, we may be updating this soon.

Completions where two or less characters are being added automatically have a score set to 0, so we can let longer, slightly lower probability completions surface.

This often happens when a user is typing out an identifier and has a few characters left in the identifier. The completion for the identifier name has less value per keystroke, but these may have the highest score because they have a 100% probability. However, such short completions are not valuable for the user to select as they do not save keystrokes.

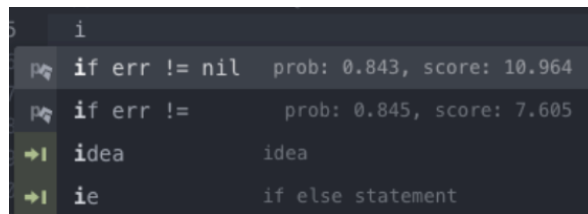
Thus, we should rank these very short completions lower than all other completions. Hiding such completions may lead to a weird experience where a user sees a completion, then as they are typing, the completion disappears. This is unexpected and could be a confusing experience.

Recommended or demoted completions

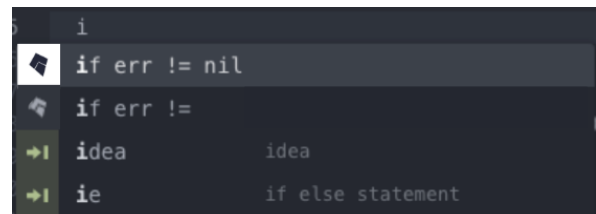
This should be in addition to [Completion Hint: \[Outdated\] Product Spec: Kite for Go](#), so should not block the work above.

Display text = Show a star.

Before



After



Some design iteration needs to happen to make it more obvious to a user about what is going on.

Similarly, we could use a different color to highlight Kite completions from other completions. However, this is not recommended due to the difficulty in implementation, as well as the variability due to a user's theme. ([link](#))

Performance Considerations

- CPU usage should never put the user in a “fan” state.
- Memory and CPU usage should be minimal when user is not actively coding in a supported language.
- Memory and CPU usage should be reduced appropriately if the user is on battery power.

Semantic integration

Using LSP to:

- identify attributes or methods, with which lexical can then provide a lengthier completion for
- infer a type for an identifier, with which, we can filter lexical completions

Custom language-specific triggers

Similar to Python, where we show completions for `import`. When we have enough usage, we can build this.

Copilot showing the right screen

When the copilot starts, restarts, or is opened from the menu bar, the screens for the right language are shown. This is determined by:

- First time after installation: Python
- Otherwise: Most recently edited language

If a user is using another completions provider

Lexical completions are designed to work in conjunction with other semantic-based completion providers. The completions list is meant to maximize completions selected, from both lexical completions and a user's semantic completion provider. We also want to reduce the effort needed to parse completions, and do so by de-duping similar completions.

Only when this is detected should we limit the number of completions returned at any given time to 2. This limit ensures that Kite's completions do not obstruct the view of semantic completions where semantic completions are valuable.

There is no differentiating between lexical or semantic providers since almost all completions provider are semantic-based at the time of writing.

Recognizing full “lines of code”

The scenarios in [De-duping Duplicate Completions](#) dedupes the # of completions that a user sees, and often removes the completion with an additional syntax token. However, a syntax token sometimes finishes the line of code for the user, and we want to show these types of completions. To do so, let’s add this new rule:

Situation 0: If one completion finishes the line of code

In this situation, the model predicts syntax tokens that can finish the line of code for the user. This is a completion that users find delight in, so we always return the completion that finishes the line of code. In this example, the model returns these completions.

```
return foo  
return foo;
```

We only show the completion that finishes the line of code.

```
return foo;
```

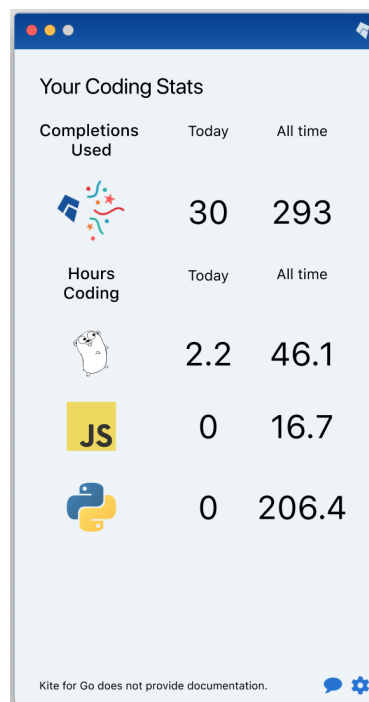
Kite Status “Warming Up”

Message	Condition
Kite: Warming up	While lexical models warm up and no Kite completions are able to be returned

Show coding stats in the Copilot

We do not have docs to show to the user so this is one idea of how to make the copilot dashboard useful to the user.

If we go this route, there should be an option for Python users to access the scorecard as well.



Always show completions when manually requested

Users who manually request completions via “Ctrl + Enter” would see the below completion, even though it’s not shown automatically as mentioned in [Line break considerations](#).

```
# Initial state
this.editor = null^
this.editor = null ^

# Display text
↵ this.editorId = uuid() # Invalid, entirety of completion goes on next line
```

Core scenarios

Almost every developer already uses an autocomplete so the best way for us to provide value is in existing scenarios that they find completions useful for. These are identified as core scenarios, and the ones that lexical should provide value for.

Identifier Name Completions

Users may not remember the exact spelling of an identifier, so they type a few characters and wait to see semantic completions that match. This is especially true if the identifier is very long or was defined by someone else. This is the same example as provided in [Baseline expectation: \[20200213 Goals\] Product Spec: Lexical Completions copy](#)

In the example below, a user has typed out a few characters with the cursor at `^` and is waiting for completions.

```
return mi^
```

Semantic completions could provide the entire identifier if it was used anywhere else in the file. The identifier in this example is especially long.

```
return minImgFileSizeInMbBase2
```

Lexical completions should provide not only the identifier, but recommend additional identifiers.

```
minImgFileSizeInMbBase2, maxImgFileSizeInMbBase2;
```

Boilerplate code

Users often retype boilerplate code, or sections of code that have to be included in many places with little or no alteration. This is distinct from the previous section of “identifiers” because much of this code can be keywords and syntax.

A semantic definition of “boilerplate” would be lines of code where consecutive tokens are of the same type, with the majority being repetitions. However, a lexical model does not know types of token.

A lexical definition would be a section of code with **two or more consecutive identifiers or keywords**, including syntax, that are exact duplicates of some code in context. The consecutive identifiers or keywords can be on separate lines.

Completions for the entire line of code show after the user has started typing the second of two consecutive identifiers or keywords. In the example below, the user would expect to see completions at the following locations:

- Line 19: `export co_`
- Line 21: `export co_`
- Line 22: `re_`
- Line 22: `then(({su_`
- Line 23: `if_`
- Line 24: `re_`
- Line 25: `ty_`
- Line 27: `da_`

For line 22: `re_`, the first of two consecutive identifiers or keywords is on line 21, at `dispatch`. This is the same for the examples from lines 23-25.

```

5  export const UPLOAD_LOGS = 'upload logs'
6
7  export const uploadLogs = () => dispatch => {
8    return dispatch(GET({url: uploadLogsPath()})).then(({success, data}) => {
9      if (success) {
10         return dispatch({
11           type: UPLOAD_LOGS,
12           success,
13           data,
14         })
15       }
16     })
17   }
18
19  export const CAPTURE = 'capture'
20
21  export const capture = () => dispatch => {
22    return dispatch(GET({url: capturePath()})).then(({success, data}) => {
23      if (success) {
24         return dispatch({
25           type: CAPTURE,
26           success,
27           data,
28         })
29       }
30     })
31   }

```

Attribute completions

Users want attribute completions to remind them what an object can do. As semantic completions provide one token that is semantically valid, lexical needs to recommend multiple identifiers that it's confident in to be valuable.

In the example below, the user has typed out a few characters with the cursor at `_` and is waiting for completions.

```
Object.
```

Some semantic completions return an ordered list of valid attributes, including this one:

```
keys
```

Lexical completions identify that in the current context, the user has been using `Object.keys` before and recommend not only that attribute, but additional identifiers as well.

```
Object.keys(fieldValues).forEach (
```

Brainstorming

Boilerplate code

Perhaps a way to see that we swapped in `CAPTURE` for `UPLOAD_LOGS`, so when we write `type: $` can we be smart enough to recommend `CAPTURE`?

- Or if we're not smart enough, recognize that it may be smart

to recommend a placeholder?

```
5  export const UPLOAD_LOGS = 'upload logs'
6
7  export const uploadLogs = () => dispatch => {
8    return dispatch(GET({url: uploadLogsPath()})).then(({success, data}) => {
9      if (success) {
10        return dispatch({
11          type: UPLOAD_LOGS,
12          success,
13          data,
14        })
15      }
16    })
17  }
18
19  export const CAPTURE = 'capture'
20
21  export const capture = () => dispatch => {
22    return dispatch(GET({url: capturePath()})).then(({success, data}) => {
23      if (success) {
24        return dispatch({
25          type: CAPTURE,
26          success,
27          data,
28        })
29      }
30    })
31  }
```

Context

Completions should take the following into context:

- Long identifiers used in the same file
- The library or framework that a user is using (eg. React vs Node.js)
- Characters immediately before and after
- Some lines of code before and after
 - If the line of code immediately before is an empty line
- Clipboard history

Potential contextual info:

- Most recently edited locations, objects/attributes
- Which other files are open

How do we use this contextual information? Should it be incorporated into the model, should it be surfaced always at the top, etc? Provide examples. If I don't know the behavior, it shouldn't be specified in the spec.

These could be leveraged in several ways:

- Recommending these long identifiers may be done by identifying the longest identifiers in the same or other currently open files, those recently typed by the user, and a user's clipboard history.
- Recommending this boilerplate code may be done by recognizing the library that a user is working in or searching through currently open files or files in the project directory.
- Recommending this attribute line of code may be done by identifying recently used objects and attributes, as well as increasing the context coverage in terms of indexed lines of code in the current file.