Pygments
Python syntax highlighter

Write your own lexer

If a lexer for your favorite language is missing in the Pygments package, you can easily write your own and extend Pygments.

All you need can be found inside the **`pygments.lexer`** module. As you can read in the *API documentation*, a lexer is a class that is initialized with some keyword arguments (the lexer options) and that provides a **`get_tokens_unprocessed()`** method which is given a string or unicode object with the data to lex.

The **`get_tokens_unprocessed()`** method must return an iterator or iterable containing tuples in the form `(index, token, value)`. Normally you don't need to do this since there are base lexers that do most of the work and that you can subclass.

RegexLexer

The lexer base class used by almost all of Pygments' lexers is the **`RegexLexer`**. This class allows you to define lexing rules in terms of *regular expressions* for different *states*.

States are groups of regular expressions that are matched against the input string at the *current position*. If one of these expressions matches, a corresponding action is performed (such as yielding a token with a specific type, or changing state), the current position is set to where the last match ended and the matching process continues with the first regex of the current state.

Lexer states are kept on a stack: each time a new state is entered, the new state is pushed onto the stack. The most basic lexers (like the *`DiffLexer`*) just need one state.

Each state is defined as a list of tuples in the form (*`regex`*, *`action`*, *`new_state`*) where the last item is optional. In the most basic form, *`action`* is a token type (like *`Name.Builtin`*). That means: When *`regex`* matches, emit a token with the match text and type *`tokentype`* and push *`new_state`* on the state stack. If the new state is `'#pop'`, the topmost state is popped from the stack instead. To pop more than one state, use `'#pop:2'` and so on. `'#push'` is a synonym for pushing the current state on the stack.

The following example shows the *`DiffLexer`* from the builtin lexers. Note that it contains some additional attributes *`name`*, *`aliases`* and *`filenames`* which aren't required for a lexer. They are used by the builtin lexer lookup functions.

```
from pygments.lexer import RegexLexer
from pygments.token import *

class DiffLexer(RegexLexer):
    name = 'Diff'
    aliases = ['diff']
    filenames = ['*.diff']

    tokens = {
        'root': [
            (r' .*\n', Text),
            (r'\+.*\n', Generic.Inserted),
            (r'-.*\n', Generic.Deleted),
```

```
            (r'@.*\n', Generic.Subheading),
            (r'Index.*\n', Generic.Heading),
            (r'=.*\n', Generic.Heading),
            (r'.*\n', Text),
        ]
    }
```

As you can see this lexer only uses one state. When the lexer starts scanning the text, it first checks if the current character is a space. If this is true it scans everything until newline and returns the data as a *Text* token (which is the "no special highlighting" token).

If this rule doesn't match, it checks if the current char is a plus sign. And so on.

If no rule matches at the current position, the current char is emitted as an *Error* token that indicates a lexing error, and the position is increased by one.

Adding and testing a new lexer

The easiest way to use a new lexer is to use Pygments' support for loading the lexer from a file relative to your current directory.

First, change the name of your lexer class to CustomLexer:

```
from pygments.lexer import RegexLexer
from pygments.token import *

class CustomLexer(RegexLexer):
    """All your lexer code goes here!"""
```

Then you can load the lexer from the command line with the additional flag `-x`:

```
$ pygmentize -l your_lexer_file.py -x
```

To specify a class name other than CustomLexer, append it with a colon:

```
$ pygmentize -l your_lexer.py:SomeLexer -x
```

Or, using the Python API:

```
# For a lexer named CustomLexer
your_lexer = load_lexer_from_file(filename, **options)

# For a lexer named MyNewLexer
your_named_lexer = load_lexer_from_file(filename, "MyNewLexer", **options)
```

When loading custom lexers and formatters, be extremely careful to use only trusted files; Pygments will perform the equivalent of `eval` on them.

If you only want to use your lexer with the Pygments API, you can import and instantiate the lexer yourself, then pass it to **`pygments.highlight()`**.

To prepare your new lexer for inclusion in the Pygments distribution, so that it will be found when passing filenames or lexer aliases from the command line, you have to perform the following steps.

First, change to the current directory containing the Pygments source code. You will need to have either an unpacked source tarball, or (preferably) a copy cloned from BitBucket.

```
$ cd .../pygments-main
```

Select a matching module under `pygments/lexers`, or create a new module for your lexer class.

Next, make sure the lexer is known from outside of the module. All modules in the `pygments.lexers` package specify `__all__`. For example, `esoteric.py` sets:

```
__all__ = ['BrainfuckLexer', 'BefungeLexer', ...]
```

Add the name of your lexer class to this list (or create the list if your lexer is the only class in the module).

Finally the lexer can be made publicly known by rebuilding the lexer mapping:

```
$ make mapfiles
```

To test the new lexer, store an example file with the proper extension in `tests/examplefiles`. For example, to test your `DiffLexer`, add a `tests/examplefiles/example.diff` containing a sample diff output.

Now you can use pygmentize to render your example to HTML:

```
$ ./pygmentize -O full -f html -o /tmp/example.html tests/examplefiles/example.diff
```

Note that this explicitly calls the `pygmentize` in the current directory by preceding it with `./`. This ensures your modifications are used. Otherwise a possibly already installed, unmodified version without your new lexer would have been called from the system search path (`$PATH`).

To view the result, open `/tmp/example.html` in your browser.

Once the example renders as expected, you should run the complete test suite:

```
$ make test
```

It also tests that your lexer fulfills the lexer API and certain invariants, such as that the concatenation of all token text is the same as the input text.

Regex Flags

You can either define regex flags locally in the regex (`r'(?x)foo bar'`) or globally by adding a *`flags`* attribute to your lexer class. If no attribute is defined, it defaults to *`re.MULTILINE`*. For more information about regular expression flags see the page about regular expressions in the Python documentation.

Scanning multiple tokens at once

So far, the *`action`* element in the rule tuple of regex, action and state has been a single token type. Now we look at the first of several other possible values.

Here is a more complex lexer that highlights INI files. INI files consist of sections, comments and `key = value` pairs:

```
from pygments.lexer import RegexLexer, bygroups
from pygments.token import *

class IniLexer(RegexLexer):
    name = 'INI'
    aliases = ['ini', 'cfg']
    filenames = ['*.ini', '*.cfg']

    tokens = {
        'root': [
            (r'\s+', Text),
            (r';.*?$', Comment),
            (r'\[.*?\]$', Keyword),
            (r'(.*?)(\s*)(=)(\s*)(.*?)$',
             bygroups(Name.Attribute, Text, Operator, Text, String))
        ]
```

```
    }
```

The lexer first looks for whitespace, comments and section names. Later it looks for a line that looks like a key, value pair, separated by an `'='` sign, and optional whitespace.

The *bygroups* helper yields each capturing group in the regex with a different token type. First the *Name.Attribute* token, then a *Text* token for the optional whitespace, after that a *Operator* token for the equals sign. Then a *Text* token for the whitespace again. The rest of the line is returned as *String*.

Note that for this to work, every part of the match must be inside a capturing group (a `(...)`), and there must not be any nested capturing groups. If you nevertheless need a group, use a non-capturing group defined using this syntax: `(?:some|words|here)` (note the `?:` after the beginning parenthesis).

If you find yourself needing a capturing group inside the regex which shouldn't be part of the output but is used in the regular expressions for backreferencing (eg: `r'(<(foo|bar)>)(.*?)(</\2>)'`), you can pass *None* to the bygroups function and that group will be skipped in the output.

Changing states

Many lexers need multiple states to work as expected. For example, some languages allow multiline comments to be nested. Since this is a recursive pattern it's impossible to lex just using regular expressions.

Here is a lexer that recognizes C++ style comments (multi-line with `/* */` and single-line with `//` until end of line):

```
from pygments.lexer import RegexLexer
from pygments.token import *

class CppCommentLexer(RegexLexer):
    name = 'Example Lexer with states'

    tokens = {
        'root': [
            (r'[^/]+', Text),
            (r'/\*', Comment.Multiline, 'comment'),
            (r'//.*?$', Comment.Singleline),
            (r'/', Text)
        ],
        'comment': [
            (r'[^*/]', Comment.Multiline),
            (r'/\*', Comment.Multiline, '#push'),
            (r'\*/', Comment.Multiline, '#pop'),
            (r'[*/]', Comment.Multiline)
        ]
    }
```

This lexer starts lexing in the `'root'` state. It tries to match as much as possible until it finds a slash (`'/'`). If the next character after the slash is an asterisk (`'*'`) the *RegexLexer* sends those two characters to the output stream marked as *Comment.Multiline* and continues lexing with the rules defined in the `'comment'` state.

If there wasn't an asterisk after the slash, the *RegexLexer* checks if it's a Singleline comment (i.e. followed by a second slash). If this also wasn't the case it must be a single slash, which is not a comment starter (the separate regex for a single slash must also be given, else the slash would be marked as an error token).

Inside the `'comment'` state, we do the same thing again. Scan until the lexer finds a star or slash. If it's the opening of a multiline comment, push the `'comment'` state on the stack and continue scanning, again in the `'comment'` state. Else, check if it's the end of the multiline comment. If yes, pop one state from the stack.

Note: If you pop from an empty stack you'll get an *IndexError*. (There is an easy way to prevent this from happening: don't `'#pop'` in the root state).

If the *RegexLexer* encounters a newline that is flagged as an error token, the stack is emptied and the lexer continues scanning in the `'root'` state. This can help producing error-tolerant highlighting for erroneous input, e.g. when a single-line string is not closed.

Advanced state tricks

There are a few more things you can do with states:

- You can push multiple states onto the stack if you give a tuple instead of a simple string as the third item in a rule tuple. For example, if you want to match a comment containing a directive, something like:

  ```
  /* <processing directive>    rest of comment */
  ```

  you can use this rule:

  ```
  tokens = {
      'root': [
          (r'/\* <', Comment, ('comment', 'directive')),
          ...
      ],
      'directive': [
          (r'[^>]*', Comment.Directive),
          (r'>', Comment, '#pop'),
      ],
      'comment': [
          (r'[^*]+', Comment),
          (r'\*/', Comment, '#pop'),
          (r'\*', Comment),
      ]
  }
  ```

  When this encounters the above sample, first `'comment'` and `'directive'` are pushed onto the stack, then the lexer continues in the directive state until it finds the closing `>`, then it continues in the comment state until the closing `*/`. Then, both states are popped from the stack again and lexing continues in the root state.

  New in version 0.9: The tuple can contain the special `'#push'` and `'#pop'` (but not `'#pop:n'`) directives.

- You can include the rules of a state in the definition of another. This is done by using *`include`* from *`pygments.lexer`*:

  ```
  from pygments.lexer import RegexLexer, bygroups, include
  from pygments.token import *

  class ExampleLexer(RegexLexer):
      tokens = {
          'comments': [
              (r'/\*.*?\*/', Comment),
              (r'//.*?\n', Comment),
          ],
          'root': [
              include('comments'),
              (r'(function )(\w+)( {)',
               bygroups(Keyword, Name, Keyword), 'function'),
              (r'.', Text),
          ],
          'function': [
              (r'[^}/]+', Text),
              include('comments'),
              (r'/', Text),
              (r'\}', Keyword, '#pop'),
          ]
      }
  ```

  This is a hypothetical lexer for a language that consist of functions and comments. Because comments can occur at toplevel and in functions, we need rules for comments in both states. As you can see, the *`include`* helper saves repeating rules that occur more than once (in this example, the state `'comment'` will never be entered by the lexer, as it's only there to be included in `'root'` and `'function'`).

- Sometimes, you may want to "combine" a state from existing ones. This is possible with the *combined* helper from *pygments.lexer*.

  If you, instead of a new state, write `combined('state1', 'state2')` as the third item of a rule tuple, a new anonymous state will be formed from state1 and state2 and if the rule matches, the lexer will enter this state.

  This is not used very often, but can be helpful in some cases, such as the *PythonLexer*'s string literal processing.

- If you want your lexer to start lexing in a different state you can modify the stack by overriding the *get_tokens_unprocessed()* method:

```
from pygments.lexer import RegexLexer

class ExampleLexer(RegexLexer):
    tokens = {...}

    def get_tokens_unprocessed(self, text, stack=('root', 'otherstate')):
        for item in RegexLexer.get_tokens_unprocessed(self, text, stack):
            yield item
```

  Some lexers like the *PhpLexer* use this to make the leading `<?php` preprocessor comments optional. Note that you can crash the lexer easily by putting values into the stack that don't exist in the token map. Also removing `'root'` from the stack can result in strange errors!

- In some lexers, a state should be popped if anything is encountered that isn't matched by a rule in the state. You could use an empty regex at the end of the state list, but Pygments provides a more obvious way of spelling that: `default('#pop')` is equivalent to `('', Text, '#pop')`.

  New in version 2.0.

Subclassing lexers derived from RegexLexer

New in version 1.6.

Sometimes multiple languages are very similar, but should still be lexed by different lexer classes.

When subclassing a lexer derived from RegexLexer, the `tokens` dictionaries defined in the parent and child class are merged. For example:

```
from pygments.lexer import RegexLexer, inherit
from pygments.token import *

class BaseLexer(RegexLexer):
    tokens = {
        'root': [
            ('[a-z]+', Name),
            (r'/\*', Comment, 'comment'),
            ('"', String, 'string'),
            ('\s+', Text),
        ],
        'string': [
            ('[^"]+', String),
            ('"', String, '#pop'),
        ],
        'comment': [
            ...
        ],
    }

class DerivedLexer(BaseLexer):
    tokens = {
        'root': [
            ('[0-9]+', Number),
            inherit,
        ],
```

```
        'string': [
            (r'[^"\\]+', String),
            (r'\\.', String.Escape),
            ('"', String, '#pop'),
        ],
    }
```

The *BaseLexer* defines two states, lexing names and strings. The *DerivedLexer* defines its own tokens dictionary, which extends the definitions of the base lexer:

- The “root” state has an additional rule and then the special object *inherit*, which tells Pygments to insert the token definitions of the parent class at that point.
- The “string” state is replaced entirely, since there is not *inherit* rule.
- The “comment” state is inherited entirely.

Using multiple lexers

Using multiple lexers for the same input can be tricky. One of the easiest combination techniques is shown here: You can replace the action entry in a rule tuple with a lexer class. The matched text will then be lexed with that lexer, and the resulting tokens will be yielded.

For example, look at this stripped-down HTML lexer:

```
from pygments.lexer import RegexLexer, bygroups, using
from pygments.token import *
from pygments.lexers.javascript import JavascriptLexer

class HtmlLexer(RegexLexer):
    name = 'HTML'
    aliases = ['html']
    filenames = ['*.html', '*.htm']

    flags = re.IGNORECASE | re.DOTALL
    tokens = {
        'root': [
            ('[^<&]+', Text),
            ('&.*?;', Name.Entity),
            (r'<\s*script\s*', Name.Tag, ('script-content', 'tag')),
            (r'<\s*[a-zA-Z0-9:]+', Name.Tag, 'tag'),
            (r'<\s*/\s*[a-zA-Z0-9:]+\s*>', Name.Tag),
        ],
        'script-content': [
            (r'(.+?)(<\s*/\s*script\s*>)',
             bygroups(using(JavascriptLexer), Name.Tag),
             '#pop'),
        ]
    }
```

Here the content of a `<script>` tag is passed to a newly created instance of a *JavascriptLexer* and not processed by the *HtmlLexer*. This is done using the *using* helper that takes the other lexer class as its parameter.

Note the combination of *bygroups* and *using*. This makes sure that the content up to the `</script>` end tag is processed by the *JavascriptLexer*, while the end tag is yielded as a normal token with the *Name.Tag* type.

Also note the `(r'<\s*script\s*', Name.Tag, ('script-content', 'tag'))` rule. Here, two states are pushed onto the state stack, `'script-content'` and `'tag'`. That means that first `'tag'` is processed, which will lex attributes and the closing `>`, then the `'tag'` state is popped and the next state on top of the stack will be `'script-content'`.

Since you cannot refer to the class currently being defined, use *this* (imported from *pygments.lexer*) to refer to the current lexer class, i.e. `using(this)`. This construct may seem unnecessary, but this is often the most obvious way of lexing arbitrary syntax between fixed delimiters without introducing deeply nested states.

The *using()* helper has a special keyword argument, *state*, which works as follows: if given, the lexer to use initially is

not in the `"root"` state, but in the state given by this argument. This does not work with advanced *`RegexLexer`* subclasses such as *`ExtendedRegexLexer`* (see below).

Any other keywords arguments passed to *`using()`* are added to the keyword arguments used to create the lexer.

Delegating Lexer

Another approach for nested lexers is the *`DelegatingLexer`* which is for example used for the template engine lexers. It takes two lexers as arguments on initialisation: a *`root_lexer`* and a *`language_lexer`*.

The input is processed as follows: First, the whole text is lexed with the *`language_lexer`*. All tokens yielded with the special type of `Other` are then concatenated and given to the *`root_lexer`*. The language tokens of the *`language_lexer`* are then inserted into the *`root_lexer`*'s token stream at the appropriate positions.

```
from pygments.lexer import DelegatingLexer
from pygments.lexers.web import HtmlLexer, PhpLexer

class HtmlPhpLexer(DelegatingLexer):
    def __init__(self, **options):
        super(HtmlPhpLexer, self).__init__(HtmlLexer, PhpLexer, **options)
```

This procedure ensures that e.g. HTML with template tags in it is highlighted correctly even if the template tags are put into HTML tags or attributes.

If you want to change the needle token `Other` to something else, you can give the lexer another token type as the third parameter:

```
DelegatingLexer.__init__(MyLexer, OtherLexer, Text, **options)
```

Callbacks

Sometimes the grammar of a language is so complex that a lexer would be unable to process it just by using regular expressions and stacks.

For this, the *`RegexLexer`* allows callbacks to be given in rule tuples, instead of token types (*`bygroups`* and *`using`* are nothing else but preimplemented callbacks). The callback must be a function taking two arguments:

- the lexer itself
- the match object for the last matched rule

The callback must then return an iterable of (or simply yield) `(index, tokentype, value)` tuples, which are then just passed through by *`get_tokens_unprocessed()`*. The `index` here is the position of the token in the input string, `tokentype` is the normal token type (like *`Name.Builtin`*), and `value` the associated part of the input string.

You can see an example here:

```
from pygments.lexer import RegexLexer
from pygments.token import Generic

class HypotheticLexer(RegexLexer):

    def headline_callback(lexer, match):
        equal_signs = match.group(1)
        text = match.group(2)
        yield match.start(), Generic.Headline, equal_signs + text + equal_signs

    tokens = {
        'root': [
            (r'(=+)(.*?)(\1)', headline_callback)
        ]
    }
```

If the regex for the *headline_callback* matches, the function is called with the match object. Note that after the callback is done, processing continues normally, that is, after the end of the previous match. The callback has no possibility to influence the position.

There are not really any simple examples for lexer callbacks, but you can see them in action e.g. in the *SMLLexer* class in ml.py.

The ExtendedRegexLexer class

The *RegexLexer*, even with callbacks, unfortunately isn't powerful enough for the funky syntax rules of languages such as Ruby.

But fear not; even then you don't have to abandon the regular expression approach: Pygments has a subclass of *RegexLexer*, the *ExtendedRegexLexer*. All features known from RegexLexers are available here too, and the tokens are specified in exactly the same way, *except* for one detail:

The *get_tokens_unprocessed()* method holds its internal state data not as local variables, but in an instance of the *pygments.lexer.LexerContext* class, and that instance is passed to callbacks as a third argument. This means that you can modify the lexer state in callbacks.

The *LexerContext* class has the following members:

- *text* – the input text
- *pos* – the current starting position that is used for matching regexes
- *stack* – a list containing the state stack
- *end* – the maximum position to which regexes are matched, this defaults to the length of *text*

Additionally, the *get_tokens_unprocessed()* method can be given a *LexerContext* instead of a string and will then process this context instead of creating a new one for the string argument.

Note that because you can set the current position to anything in the callback, it won't be automatically be set by the caller after the callback is finished. For example, this is how the hypothetical lexer above would be written with the *ExtendedRegexLexer*:

```
from pygments.lexer import ExtendedRegexLexer
from pygments.token import Generic

class ExHypotheticLexer(ExtendedRegexLexer):

    def headline_callback(lexer, match, ctx):
        equal_signs = match.group(1)
        text = match.group(2)
        yield match.start(), Generic.Headline, equal_signs + text + equal_signs
        ctx.pos = match.end()

    tokens = {
        'root': [
            (r'(=+)(.*?)(\1)', headline_callback)
        ]
    }
```

This might sound confusing (and it can really be). But it is needed, and for an example look at the Ruby lexer in ruby.py.

Handling Lists of Keywords

For a relatively short list (hundreds) you can construct an optimized regular expression directly using `words()` (longer lists, see next section). This function handles a few things for you automatically, including escaping metacharacters and Python's first-match rather than longest-match in alternations. Feel free to put the lists themselves in `pygments/lexers/_$lang_builtins.py` (see examples there), and generated by code if possible.

An example of using `words()` is something like:

```
from pygments.lexer import RegexLexer, words, Name

class MyLexer(RegexLexer):

    tokens = {
        'root': [
            (words(('else', 'elseif'), suffix=r'\b'), Name.Builtin),
            (r'\w+', Name),
        ],
    }
```

As you can see, you can add `prefix` and `suffix` parts to the constructed regex.

Modifying Token Streams

Some languages ship a lot of builtin functions (for example PHP). The total amount of those functions differs from system to system because not everybody has every extension installed. In the case of PHP there are over 3000 builtin functions. That's an incredibly huge amount of functions, much more than you want to put into a regular expression.

But because only *Name* tokens can be function names this is solvable by overriding the `get_tokens_unprocessed()` method. The following lexer subclasses the *PythonLexer* so that it highlights some additional names as pseudo keywords:

```
from pygments.lexers.python import PythonLexer
from pygments.token import Name, Keyword

class MyPythonLexer(PythonLexer):
    EXTRA_KEYWORDS = set(('foo', 'bar', 'foobar', 'barfoo', 'spam', 'eggs'))

    def get_tokens_unprocessed(self, text):
        for index, token, value in PythonLexer.get_tokens_unprocessed(self, text):
            if token is Name and value in self.EXTRA_KEYWORDS:
                yield index, Keyword.Pseudo, value
            else:
                yield index, token, value
```

The *PhpLexer* and *LuaLexer* use this method to resolve builtin functions.