

---

# **TDParser Documentation**

*Release 1.1.6*

**Raphaël Barrois**

**May 24, 2017**



---

# Contents

---

<b>1</b>	<b>Getting started</b>	<b>1</b>
1.1	Installation . . . . .	1
1.2	Defining the tokens . . . . .	1
1.3	Building the Lexer/Parser . . . . .	2
<b>2</b>	<b>Contents</b>	<b>3</b>
2.1	Reference . . . . .	3
2.2	TDParser internals . . . . .	7
2.3	ChangeLog . . . . .	9
<b>3</b>	<b>Links</b>	<b>11</b>
3.1	TDParser resources . . . . .	11
3.2	External references . . . . .	11
<b>4</b>	<b>Indices and tables</b>	<b>13</b>
	<b>Python Module Index</b>	<b>15</b>



### Installation

First, you'll need to get the latest version of TDParse.

TDParse is compatible with all Python versions from 2.6 to 3.2.

The simplest way is to get it from PyPI:

```
$ pip install tdpaser
```

You may also fetch the latest development version from <https://github.com/rbarrois/tdpaser>:

```
$ git clone git://github.com/rbarrois/tdpaser.git
$ cd parser
$ python setup.py install
```

### Defining the tokens

TDParse provides a simple framework for building parsers; thus, it doesn't provide default token kinds.

Defining a token type requires 4 elements:

- Input for the token: TDParse uses a regexp, in the `tdpaser.Token.regexp` attribute
- Precedence, an integer stored in the `tdpaser.Token.lbp` attribute
- Value that the token should get when it appears at the beginning of a standalone expression; this behavior is defined in the `tdpaser.Token.nud()` method
- Behavior of the token when it appears between two expressions; this is defined in the `tdpaser.Token.lod()` method.

An example definition of a simple arithmetic parser that returns the expression's value would be:

```
from tdparsor import Token

class Integer(Token):
    regexp = r'\d+'
    def nud(self, context):
        return int(self.text)

class Addition(Token):
    regexp = r'\+'
    lbp = 10

    def led(self, left, context):
        return left + context.expression(self.lbp)

class Multiplication(Token):
    regexp = r'\*'
    lbp = 20

    def led(self, left, context):
        return left * context.expression(self.lbp)
```

## Building the Lexer/Parser

The parser has a simple interface: it takes as input an iterable of *tdparsor.Token*, and returns the expression that the tokens' *nud()* and *led()* methods return.

The lexer simply needs to get a list of valid tokens:

```
lexer = tdparsor.Lexer(with_parens=True)
lexer.register_tokens(Integer, Addition, Multiplication)
```

The *with\_parens=True* options adds a pair of builtin tokens, *tdparsor.LeftParen* and *tdparsor.RightParen*, which provide left/right parenthesis behavior.

---

**Note:** The default lexer will skip space and tabulations. This can be modified by settings the *blank\_chars* argument when initializing the lexer.

---

We now only need to feed our text to the lexer:

```
>>> lexer.parse('1 + 1')
2
>>> lexer.parse('2 * 3 + 4')
10
```

## Reference

This document describes all components of the `tdparser` package:

- *Token*
- *Parser*
- *Lexer*

## Exception classes

**exception** `tdparser.Error`

This exception is the base class for all `tdparser`-related exceptions.

**exception** `tdparser.ParserError` (*Error*)

This exception will be raised whenever an unexpected token is encountered in the flow of tokens.

**exception** `tdparser.MissingTokensError` (*ParserError*)

This exception is raised when the parsing logic would expect more tokens than are available

**exception** `tdparser.InvalidTokenError` (*ParserError*)

This exception is raised when an unexpected token is encountered while parsing the data flow.

## Defining tokens

A token must inherit from the *Token* class, and override a few elements depending on its role.

**class** `tdparser.Token`

The base class for all tokens.

**regex**

Class attribute.

Optional regular expression (see `re`) describing text that should be lexed into this token class.

**Type** `str`

**lbp**

Class attribute.

“Left binding power”. This integer describes the precedence of the token when it stands at the left of an expression.

Tokens with a higher binding power will absorb the next tokens in priority:

In `1 + 2 * 3 + 4`, if `+` has a `lbp` of 10 and `*` of 20, the `2 * 3` part will be computed and its result passed as a right expression to the first `+`.

**Type** `int`

**text**

The text that matched `regex`.

**Type** `str`

**nud** (*self*, *context*)

Compute the “Null denotation” of this token.

This method should only be overridden for tokens that may appear at the beginning of an expression.

For instance, a number, a variable name, the “-” sign when denoting “the opposite of the next expression”.

The `context` argument is the `Parser` currently running. This gives easy access to:

- The next token in the flow (`Parser.current_token`)
- The position in the flow of tokens (`Parser.current_pos`)
- Retrieving the next sub-expression from the parser (`Parser.expression()`)

**Parameters** `context` (`tdparser.Parser`) – The active `Parser`

**Returns** The value this token evaluates to

**led** (*self*, *left*, *context*)

Compute the “Left denotation” of this token.

This method is called whenever a token appears to the right of another token within an expression — typically infix or postfix operators.

It receives two arguments:

- `left` is the value of the previous token or expression in the flow
- `context` is the active `Parser` instance, providing calls to `Parser.expression()` to fetch the next expression.

**Parameters**

- **left** – Whatever the previous expression evaluated to
- **context** (`tdparser.Parser`) – The active `Parser`

**Returns** The value this token evaluates to

**class** `tdparser.LeftParen` (*Token*)

A simple `Token` subclass matching an opening bracket, `(`.



When parsed, this token will fetch the next subexpression, assert that this expression is followed by a *RightParen* token, and return the value of the fetched expression.

**match**

The token class to expect at the end of the subexpression. This simplifies writing similar “bracket” tokens with different opening/closing signs.

**Type** *Token*

**class** `tdparser.RightParen` (*Token*)

A simple, passive *Token* (returns no value).

Used by the *LeftParen* token to check that the sub-expression was properly enclosed in left/right brackets.

**class** `tdparser.EndToken` (*Token*)

This specific *Token* marks the end of the input stream.

## Parsing a flow of tokens

The actual parsing occurs in the *Parser* class, which takes a flow of *Token*.

Parsing is performed through the *parse()* method, which will return the next parsed expression.

**class** `tdparser.Parser`

Handles parsing of a flow of tokens. Maintains a pointer to the current *Token*.

**current\_pos**

Stores the current position within the token flow. Starts at 0.

**Type** `int`

**current\_token**

The next *Token* to parse. When calling a token’s *nud()* or *led()*, this attribute points to the *next* token, not the token whose method has been called.

**Type** *Token*

**tokens**

Iterable of tokens to parse. Can be any kind of iterable — will only be walked once.

**Type** iterable of *Token*

**consume** (*self*, *expect\_class=None*)

Consume the active *current\_token*, and advance to the next token.

If the *expect\_class* is provided, this will ensure that the *current\_token* matches that token class, and raise a *InvalidTokenError* otherwise.

**Parameters** *expect\_class* (`tdparser.Token`) – Optionnal *Token* subclass that the *current\_token* should be an instance of

**Returns** the *current\_token* at the time of calling.

**expression** (*self*, *rbp=0*)

Retrieve the next expression from the flow of tokens.

The *rbp* argument describes the “right binding power” of the calling token. This means that the parsing of the expression will stop at the first token whose left binding power is lower than this right binding power.

This obscure definition describes the right precedence of a token. In other words, it means “fetch an expression, and stop whenever you meet an operator with a lower precedence”.

## Example

In the `1 + 2 * 3 ** 4 + 5`, the `led()` method of the `*` token will call `context.expression(20)`. This call will absorb the `3 ** 4` part as a single expression, and stop when meeting the `+`, whose left binding power, 10, is lower than 20.

**Parameters** `rbp` (*int*) – The (optional) right binding power to use when fetching the next subexpression.

**parse** (*self*)

Compute the first expression from the flow of tokens.

## Generating tokens from a string

The `Parser` class works on an iterable of `tokens`.

In order to retrieve those tokens, the simplest way is to use the `Lexer` class.

**class** `tdparser.Lexer`

This class handles converting a string into an iterable of `tokens`.

Once initialized, a `Lexer` must be passed a set of tokens to handle.

The lexer parses strings according to the following algorithm:

- Try each regexp in order for a match at the start of the string
- If none match:
  - If the first character is a blank (see `blank_chars`), remove it from the beginning of the string and go back to step 1
  - Otherwise, raise a `ValueError`.
- If more than one regexp match, keep the one with the longest match.
- Among those with the same, longest, match, keep the first registered one
- Instantiate the `Token` associated with that best regexp, passing its constructor the substring that was matched by the regexp
- Yield that `Token` instance
- Strip the matched substring from the text, and go back to step 1.

---

**Note:** The `Lexer` can be used as a standalone parser: the tokens passed to `Lexer.register_token()` are simply instantiated with the matching text as first argument.

---

**tokens**

A `TokenRegistry` holding the set of known tokens.

**Type** `TokenRegistry`

**blank\_chars**

An iterable of chars that should be considered as “blank” and thus not parsed into a `Token`.

**Type** iterable of `str`

**end\_token**

The `Token` subclass to use to mark the end of the flow

**Type** `EndToken`

**register\_token** (*self*, *token\_class* [, *regex*=None ])

Registers a token class in the lexer (actually, in the *TokenRegistry* at *tokens*).

There are two methods to provide the regular expression for token extraction:

- In the *regex* parameter to *register\_token* ()
- If that parameter isn't provided, the *Lexer* will look for a *regex* string attribute on the provided *token\_class*.

#### Parameters

- **token\_class** (*tdparser.Token*) – The *Token* subclass to add to the list of available tokens
- **regex** (*str*) – The regular expression to use when extracting tokens from some text; if empty, the *regex* attribute of the *token\_class* will be used instead.

**register\_tokens** (*self*, *token\_class* [, *token\_class* [, ... ] ])

Register a batch of *Token* subclasses. This is equivalent to calling *lexer.register\_token* (*token\_class*) for each passed *token\_class*.

The regular expression associated to each token *must* be set on its *regex* attribute; no overrides are available with this method.

**Parameters** **token\_class** (*tdparser.Token*) – token classes to register

**lex** (*self*, *text*)

Read a text, and lex it, yielding *Token* instances.

This will walk the text, eating chunks that can be paired to a *Token* through its associated regular expression.

It will yield *Token* instances while parsing the text, and end with an instance of the *EndToken* class as set in the *lexer*'s *end\_token* attribute.

**Parameters** **text** (*str*) – The text to lex

**Returns** Iterable of *Token* instances

**parse** (*self*, *text*)

Shortcut method for lexing and parsing a text.

Will *lex* () the text, then instantiate a *Parser* with the resulting *Token* flow and call its *parse* () method.

## TDParse internals

Beyond the *Reference* section, here is an in-depth description of *tdparser*'s internals.

### Lexer helpers

This module holds the *tdparser.Lexer* class, which is available in the top-level *tdparser* module.

**class** *tdparser.lexer.TokenRegistry*

This class holds a set of (token, regex) pairs, and selects the appropriate pair to extract data from a string.

---

**Note:** The *TokenRegistry* doesn't interact with the *Token* subclasses provided through *register* ().

This means that any kind of value could be provided for this field, and will be returned as-is by the `get_token()` method.

---

### `__tokens`

Holds a list of `(Token, re.RegexObject)` tuples. These are the tokens in the order they were inserted (insertion order matters).

**Type** list of `(Token subclass, re.RegexObject)` tuples

### `register(self, token, regexp)`

Register a `Token` subclass for the given `regexp`.

#### Parameters

- **token** (`tdparser.Token`) – The `Token` subclass to register
- **regexp** (`str`) – The regular expression (as a string) associated with the token

### `matching_tokens(self, text[, start=0])`

Retrieve all tokens matching a given text. The optional `start` argument can be used to alter the start position for the `match()` call.

#### Parameters

- **text** (`str`) – Text for which matching `(Token, re.MatchObject)` pairs should be searched
- **start** (`int`) – Optional start position with `text` for the `regexp match()` call

**Returns** Yields tuples of `(Token, re.MatchObject)` for each token whose `regexp` matched the `text`.

### `get_token(self, text[, start=0])`

Retrieve the best token class and the related `match` at the start of the given `text`.

The algorithm for choosing the “best” class is:

- Fetch all matching tokens (through `matching_tokens()`)
- Select those with the longest match
- Return the first of those tokens

A different starting position for `match()` calls can be provided in the `start` parameter.

#### Parameters

- **text** (`str`) – Text for which the `(Token, re.MatchObject)` pair should be returned
- **start** (`int`) – Optional start position with `text` for the `regexp match()` call

#### Returns

`(Token, re.MatchObject)` pair, the best match for the given `text`.

If no token matches the text, returns `(None, None)`.

### `__len__(self)`

The `len()` of a `TokenRegistry` is the length of its `__tokens` attribute.

## ChangeLog

### 1.2.0 (planned)

*New:*

- Batteries included (provide ready-to-use tokens for arithmetic evaluation, AST building, ...)
- Add documentation for the top-down algorithm

### 1.1.6 (2013-09-14)

*Misc:*

- Switch back to setuptools.

### 1.1.5 (2013-05-20)

*Bugfix:*

- #8: Fix packaging: stop installing the tests/ dir

### 1.1.4 (2013-03-11)

- Fix handling of empty token flows
- More descriptive errors

### 1.1.3 (2012-11-02)

*Bugfix:*

- Fix setup.py (`find_packages()` was installing tests as well)

### 1.1.2 (2012-11-02)

*Bugfix:*

- Swap doc/changelog.rst and ChangeLog for proper sdist compatibility

### 1.1.1 (2012-11-02)

*Bugfix:*

- Fix documentation packaging (invalid paths in MANIFEST.in)

### 1.1.0 (2012-08-24)

*New:*

- Simpler token registration to the *Lexer*
- Improved documentaion
- Full test coverage
- Python3 compatibility

*Bugfix:*

- Choose token by longest match

### 1.0.0 (2012-08-21)

First stable version of tdpaser.

Includes:

- Fully functional top-down parser
- Lexer
- Wide set of unit tests

### TDParser resources

- Project home page on GitHub: <https://github.com/rbarrois/tdparser>
- Documentation hosted on ReadTheDocs: <http://tdparser.readthedocs.org/>
- Continuous integration on Travis-CI: <http://travis-ci.org/rbarrois/tdparser>

### External references

A few articles on Top-Down parsing (the algorithm used by tdparser):

- <http://effbot.org/zone/simple-top-down-parsing.htm> : A simple implementation in Python
- <http://javascript.crockford.com/tdop/tdop.html> : A full tutorial in JS
- <http://dl.acm.org/citation.cfm?id=512931> : Original description of the algorithm





## CHAPTER 4

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**t**

`tdparser`, 3

`tdparser.lexer`, 7



## Symbols

`__len__()` (tdparser.lexer.TokenRegistry method), 8  
`_tokens` (tdparser.lexer.TokenRegistry attribute), 8

## B

`blank_chars` (tdparser.Lexer attribute), 6

## C

`current_pos` (tdparser.Parser attribute), 5  
`current_token` (tdparser.Parser attribute), 5

## E

`end_token` (tdparser.Lexer attribute), 6  
`EndToken` (class in tdparser), 5  
`Error`, 3

## G

`get_token()` (tdparser.lexer.TokenRegistry method), 8

## I

`InvalidTokenError`, 3

## L

`lbp` (tdparser.Token attribute), 4  
`led()` (tdparser.Token method), 4  
`LeftParen` (class in tdparser), 4  
`lex()` (tdparser.Lexer method), 7  
`Lexer` (class in tdparser), 6

## M

`match` (tdparser.LeftParen attribute), 5  
`matching_tokens()` (tdparser.lexer.TokenRegistry  
method), 8  
`MissingTokensError`, 3

## N

`nud()` (tdparser.Token method), 4

## P

`parse()` (tdparser.Lexer method), 7  
`Parser` (class in tdparser), 5  
`Parser.consume()` (in module tdparser), 5  
`Parser.expression()` (in module tdparser), 5  
`Parser.parse()` (in module tdparser), 6  
`ParserError`, 3

## R

`regexp` (tdparser.Token attribute), 3  
`register()` (tdparser.lexer.TokenRegistry method), 8  
`register_token()` (tdparser.Lexer method), 6  
`register_tokens()` (tdparser.Lexer method), 7  
`RightParen` (class in tdparser), 5

## T

`tdparser` (module), 3  
`tdparser.lexer` (module), 7  
`text` (tdparser.Token attribute), 4  
`Token` (class in tdparser), 3  
`TokenRegistry` (class in tdparser.lexer), 7  
`tokens` (tdparser.Lexer attribute), 6  
`tokens` (tdparser.Parser attribute), 5