partpy Documentation

Release 1.2.4

Taylor "Nekroze" Lawson

August 13, 2013

CONTENTS

Parser Tools in Python (`partpy`, pronounced `Par-Tee-Pie`), a collection of tools for hand writing lexers and parsers in python.

There are many parser generators but there isn't much help for those who wish to roll their own parser/lexer as counter-intuitive as that may sound. `partpy` provides a solid base for hand written parsers and lexers through a library of common tools.

By using `partpy` as the base for your own parser or lexer the hope is to provide you with an environment where you can dive straight into the language design, recognition and whatever else you need to do without having to figure out how string matching should be done or most of the error handling process.

Contents:

CHAPTER

ONE

TUTORIAL

The main thing that you will use when working with partpy is the partpy.sourcestring.SourceString object. While this object can be instantiated alone it is recommended to use it as a base class to inherit your own lexer/parser from.

The SourceString can take a file or a string and will store it internally along with; its length, the current index of the string, the current line and column position and if the end of the string has been reached yet.

SourceString Also has a variety of methods used for things such as; moving the current position, matching strings or string/function patterns, counting indentations and a few other useful things.

1.1 Movement

When using a SourceString it can automattically keep track of which column and line you are on in the text file as well as which index in the string it is currently operating on.

The most simple way to check a character and move around a SourceString derived object is with SourceString.get_char() and SourceString.eat_length(), respectively. .get_char() will simply return the character at the current position of the SourceString and .eat_length() will move over it to the next character.

We can use SourceString.has_space(), or to avoid function call overhead, SourceString.eos to start a loop that can keep going until broken or the entirety of the SourceString stored string has been passed. .. testcode:

```
from partpy import SourceString

class Number(SourceString):
    digits = '0123456789'
    def spew_everything():
        while not self.eos:
            char = self.get_char()
            if char in self.digits:
                yield char
            self.eat_length()

parser = Number("123abc456")
print(str(parser.spew_everything()) == '123456')
```

This class, when given a string to work with or a file, will go over every character and yield only the numbers in it. While this example is trivial and rather useless on its own it does teach us some handy things for later.

By using the self.eat_length() method, inherited from SourceString, it will automatically move the current position forward by the integer value given to self.eat_length() which is by default 1. This will handle newline characters and as such eating a length of 1 will move the SourceString position forwards by one along with the current column. However if the current character is a newline then the column is set to 0 and the current line is incremented by one.

It will always be import to eat the length of your match once you want to move past it because all SourceString matching and retrieving methods use the internally tracked positions.

1.2 Simple String Matching

There are several ways to match strings, The most explicit way is to specifically define each posible string to match.

SourceString.match_string will attempt to match a single string at the current position. SourceString.match_any_string does much the same thing but takes a list of strings and will return the string that it matches and an empty string if there is no match. There are the accompanying method; SourceString.match_any_char are much the same as the string version but takes a string of one or more characters to match against rather then a list. .. testcode:

```
from partpy import SourceString

class Parser(SourceString):
    def match():
        match = self.match_any_string(['def', 'class'])
        self.eat_length(match)

        if not match:
            return match
        elif match == 'class':
            return 'TOKEN_CLASS'
        elif match == 'def':
            return 'TOKEN_DEF'

parser = Parser('class')
print(parser.match() == 'TOKEN_CLASS')
```

In an easy and fast way we can match any specific string or character however we wish.

1.3 Pattern String Matching

SourceString also has mutltiple methods to help with string and pattern matching. For example you can match a single string or a pattern using the following. Just to simplify the example code SourceString will directly instanced. .. testcode:

```
from partpy import SourceString

myMatcher = SourceString()
myMatcher.set_string('partpy is cool')
match = myMatcher.match_string('cool')
if not match:
    match = myMatcher.match_function(str.isalpha)
print(match == 'partpy')
```

SourceString can match text in a few ways out of the box. SourceString.match_string will attempt to match from the current position (the very start at the moment because we haven't eaten anything yet) to the length of the given string and will return an empty string if nothing was found. As it will be here.

Because nothing was matched we couldn't match 'cool' at the current position we will use SourceString.match_function instead. This method can take a function that expects a single string or character argument and returns anything that can be evaluated as a boolean. We will use the builtin str.isalpha method that will return True for any alphabetical character or string.

SourceString.match_function will go from the current position forwards through the SourceString until its function does not match anymore and return the results.

There is another method, SourceString.match_pattern, which works exactly the same as SourceString.match_function but takes strings rather then functions, this means that you can re-write the previous example as. .. testcode:

```
from partpy import SourceString

myMatcher = SourceString()
myMatcher.set_string('partpy is cool')
match = myMatcher.match_string('cool')
if not match:
    match = myMatcher.match_pattern('abcdefghijklmnopqrstuvwxyz')
print(match == 'partpy')
```

This will work exactly the same and may even be faster as you can avoid function overhead when using your own functions for SourceString.match_function however there are many builtin str methods that are very useful and are much faster then your own python interpreted functions.

Both SourceString.match_function and SourceString.match_pattern can actually take two arguments. If a second argument given then the first argument is used only to match the first character and all following characters are matched using the second. This is useful for detecting 'Title' cased words for example. .. testcode:

```
from partpy import SourceString

myMatcher = SourceString()
myMatcher.set_string('Partpy is cool')
match = myMatcher.match_function(str.isupper, str.islower)
print(match == 'Partpy')
```

The two arguments may also be given as a tuple or list to the first argument only and will be unpacked into the first and second arguments automatically.

1.4 Your Implementation

As previously stated partpy was designed to be subclassed and used in your own implementations of hand written parsers and lexical analyzers. .. testcode:

```
from partpy import SourceString

class WordCollector(SourceString):
    def words(self):
        while not self.eos:
            while self.get_char().isspace():
                self.eat_string(self.get_char())
            word = self.get_string()
            self.eat_string(word)
            yield word

myCollector = WordCollector()
myCollector.set_string('these are all words')
```

```
words = [word for word in myCollector.words()]
print(words == [’these’, ’are’, ’all’, ’words’])
```

This may be a pointless example in terms of its actual usefulness but ignore that and just see how the SourceString is used rather then what this whole thing does. One can see how they can make a simple OOP class that can parse or provide lexical analyses using partpy in a very simple way.

1.5 Exceptions

Another useful thing that one should consider using is the handy PartpyError which is an exception that can be raised with a custom message and a SourceString derived object. Using this info when the exception is raised will, by default, add to the end of a python stacktrace a numbered list of the current line (and the previous one if available), aswell as a carrot underneath the current character, based on the SourceString current position. Finally it will output the custom message if defined.

```
>>>from partpy import SourceString, PartpyError
>>>source = SourceString(’Let’s use partpy’)
>>>source.eat_length(6)
>>>raise PartpyError(source, ’you broke it!’)
Traceback (most recent call last):
partpy.partpyerror.PartpyError:
1   |Let’s use partpy
           ^
you broke it!
```

CHAPTER

TWO

PERFORMANCE

partpy is written with Cython support through .pxd files and can be compiled for extra speed. If you do not want to use Cython simply install it from the source with Cython uninstalled.

If however you use pip or easy_install or something similar Cython is marked a dependency and must be installed.

partpy is tested without compilation on python{2.6, 2.7, 3.2} and the latest pypy stable release. All these platforms are also tested with cython compilation except for python{2.6}. As an additional bonus partpy is also tested for rpython translation. This means that the pypy rpython translation toolchain can compile partpy making it useable in very fast interpreters/VM's using pypy toolchain that can also provide a JIT compiler for free.

All of the compilation and usage options are designed to allow for maximum flexability while also allowing maximum performance and usage.

CHAPTER

THREE

CHANGELOG

V1.2.3

- Fix for new no space guards on patern matching not passing offsets

V1.2.2

- Added Impyccable to test suite for better pattern testing
- Added no space guards to pattern matching and indenter

V1.2.1

- Fixed .pxd files not being included in source distribution

V1.2.0

- Added Offset arguments to most SourceString methods that should support it.
- Adde '_' to qualified identifiers

v1.1.0

- Added SourceString.eol_distance_next
- Added SourceString.eol_distance_last
- Added SourceString.spew_length a reverse of eat_length
- Minor failsafes

v1.0.0

- Added RPython compatability.
- Removed some dynamic features
- Removed SourceString.generator
- SourceString.match_pattern renamed to match_string_pattern
- SourceString.match_function renamed to match_function_pattern
- Pattern matching methods only take their respective types, no more lists.

v0.3.0

- **Added SourceString methods:**
  - eat_line
  - count_indents_last_line
  - count_indents_length_last_line

    - skip_whitespace
    - get_all_lines
    - retrieve_tokens
- Added least argument to SourceString.match_(pattern/function) for minimum length of match
- SourceString.eat_length now handles newlines automatically
- Some source code cleanups and cython fixes/optimizations
- All SourceString.eat_* methods nolonger function when SourceString.eos = 1
- Added sphinx based documentation system and http://partpy.readthedocs.org
- Line numebers start at line 1

v0.2.1 - February 14th 2013

- Added examples directory to sdist

v0.2.0 - February 14th 2013

- Matcher merged into SourceString
- new class SourceLine returned when dealing with specific SourceString lines

CHAPTER

FOUR

PARTPY PACKAGE

4.1 sourcestring Module

SourceString stores the entire string to be parsed in memory and provides some simple methods for retrieving and moving current position aswell as methods for matching strings and patterns.

class partpy.sourcestring.**SourceLine**(*string*, *lineno*)

Bases: partpy.sourcestring.SourceString

Contains an entire line of a source with handy line specific methods.

get_first_char()

Return the first non-whitespace character of the line.

get_last_char()

Return the last non-whitespace character of the line.

pretty_print(*carrot=False*)

Return a string of this line including linenumber. If carrot is True then a line is added under the string with a carrot under the current character position.

strip_trailing_ws()

Remove trailing whitespace from internal string.

class partpy.sourcestring.**SourceString**(*string=None*)

Bases: object

Stores the parse string and its length followed by current position in the string and if the end of the string has been reached.

It also stores the current row and column position as manually counted.

Provides multiple methods for matching strings and patterns and working with the source string.

__contains__(*string*)

Returns a boolean if the given string is within the base string. Called by 'word' in SourceString.

__getitem__(*index*)

Returns the character at the given index. Called by SourceString[index] where index is an integer.

__init__(*string=None*)

Accepts a string or None by default. If a string is given then self.set_string(string) is run automatically. If you wish to load a file then create a SourceString object with no arguments and then use load_file or overload this function when inheriting from SourceString.

__iter__()

Yields the current char and moves the position onwards until eos.

__len__()
Returns the length of base string. Called by len(SourceString).

__repr__()
Returns the entire base string. Called from the repr() builtin.

__weakref__
list of weak references to the object (if defined)

add_string(*string*)
Add to the working string and its length and reset eos.

count_indents(*spacecount*, *tabs=0*, *offset=0*)
Counts the number of indents that can be tabs or spacecount number of spaces in a row from the current line.

count_indents_last_line(*spacecount*, *tabs=0*, *back=5*)
Finds the last meaningful line and returns its indent level. Back specifies the amount of lines to look back for a none whitespace line.

count_indents_length(*spacecount*, *tabs=0*, *offset=0*)
Counts the number of indents that can be tabs or spacecount number of spaces in a row from the current line.

Also returns the character length of the indents.

count_indents_length_last_line(*spacecount*, *tabs=0*, *back=5*)
Finds the last meaningful line and returns its indent level and character length. Back specifies the amount of lines to look back for a none whitespace line.

eat_length(*length*)
Move current position forward by length and sets eos if needed.

eat_line()
Move current position forward until the next line.

eat_string(*string*)
Move current position by length of string and count lines by .

eol_distance_last(*offset=0*)
Return the ammount of characters until the last newline.

eol_distance_next(*offset=0*)
Return the amount of characters until the next newline.

get_all_lines()
Return all lines of the SourceString as a list of SourceLine's.

get_char(*offset=0*)
Return the current character in the working string.

get_current_line()
Return a SourceLine of the current line.

get_length(*length*, *trim=0*, *offset=0*)
Return string at current position + length. If trim == true then get as much as possible before eos.

get_line(*lineno*)
Return any line as a SourceLine and None if lineno doesnt exist.

get_lines(*first*, *last*)
Return SourceLines for lines between and including first & last.

get_string (*offset=0*)
Return non space chars from current position until a whitespace.

get_surrounding_lines (*past=1*, *future=1*)
Return the current line and x,y previous and future lines. Returns a list of SourceLine's.

has_space (*length=1*, *offset=0*)
Returns boolean if self.pos + length < working string length.

load_file (*filename*)
Read in file contents and set the current string.

match_any_char (*chars*, *offset=0*)
Match and return the current SourceString char if its in chars.

match_any_string (*strings*, *word=0*, *offset=0*)
Attempts to match each string in strings in order. Will return the string that matches or an empty string if no match.

If word arg >= 1 then only match if string is followed by a whitespace which is much higher performance.

If word is 0 then you should sort the strings argument yourself by length.

match_function_pattern (*first*, *rest=None*, *least=1*, *offset=0*)
Match each char sequentially from current SourceString position until the pattern doesnt match and return all maches.

Integer argument least defines and minimum amount of chars that can be matched.

This version takes functions instead of string patterns. Each function must take one argument, a string, and return a value that can be evauluated as True or False.

If rest is defined then first is used only to match the first arg and the rest of the chars are matched against rest.

match_string (*string*, *word=0*, *offset=0*)
Returns 1 if string can be matches against SourceString's current position.

If word is >= 1 then it will only match string followed by whitepsace.

match_string_pattern (*first*, *rest=None*, *least=1*, *offset=0*)
Match each char sequentially from current SourceString position until the pattern doesnt match and return all maches.

Integer argument least defines and minimum amount of chars that can be matched.

If rest is defined then first is used only to match the first arg and the rest of the chars are matched against rest.

reset_position ()
Reset all current positions.

rest_of_string (*offset=0*)
A copy of the current position till the end of the source string.

set_string (*string*)
Set the working string and its length then reset positions.

skip_whitespace (*newlines=0*)
Moves the position forwards to the next non newline space character. If newlines >= 1 include newlines as spaces.

spew_length (*length*)
Move current position backwards by length.

4.2 partpyerror Module

Custom exception for classes inheriting SourceString or Matcher.

exception `partpy.partpyerror.`**`PartpyError`**(*obj*, *msg=None*)

Bases: `exceptions.Exception`

Takes a SourceString or Matcher derived object and an optional message.

When converted to a string will display the previous and current line with line numbers and a '^' under the current position of the object with the optional message on the following line.

`__weakref__`

list of weak references to the object (if defined)

`pretty_print`(*carrot=True*)

Print the previous and current line with line numbers and a carret under the current character position.

Will also print a message if one is given to this exception.

4.3 spattern Module

Predefined string patterns for use in Matcher.match_pattern methods.

Defines the following:

- alphal = lower case alphabet
- alphau = upper case alphabet
- alpha = lower and upper case alphabet
- number = digits
- alnum = digits or lower and upper case alphabet
- identifier = first(alpha) rest(alnum | '_')
- qualified = first(alpha) rest(alnum | '.' | '_')
- integer = first(number | '-') rest(number)

4.4 fpattern Module

Predefined function patterns for use in Matcher.match_function methods.

Defines the following:

- alphal = lower case alphabet
- alphau = upper case alphabet
- alpha = lower and upper case alphabet
- number = digits
- alnum = digits or lower and upper case alphabet
- identifier = first(alpha) rest(alnum | '_')
- qualified = first(alpha) rest(alnum | '.' | '_')

- integer = first(number | '-') rest(number)

CHAPTER

FIVE

EXAMPLES PACKAGE

This package is a series of self contained examples that use partpy to accomplish their goal. All complete examples are included in the test suite to check their output and that they function.

5.1 `contacts` Module

SourceString utilizes SourceString to provide some simple string matching functionality.

class `examples.contacts.`**`ContactsParser`**`(`*string=None*`)`

Bases: `partpy.sourcestring.SourceString`

The contacts parser will simply look for name and website pairs while disregarding any kind of whitespace and store them in a dict.

`parse``()`

Run the parser over the entire sourestring and return the results.

`parse_contact``()`

Parse a top level contact expression, these consist of a name expression a special char and an email expression.

The characters found in a name and email expression are returned.

`parse_email``()`

Email address parsing is done in several stages. First the name of the email use is determined. Then it looks for a '@' as a delimiter between the name and the site. Lastly the email site is matched.

Each part's string is stored, combined and returned.

`parse_name``()`

This function uses string patterns to match a title cased name. This is done in a loop until there are no more names to match so as to be able to include surnames etc. in the output.

`parse_top_level``()`

The top level parser will do a loop where it looks for a single contact parse and then eats all whitespace until there is no more input left or another contact is found to be parsed and stores them.

`parse_whitespace``()`

This function simply eats chars until the current char is no longer a space, tab, newline.

CHAPTER

SIX

FEEDBACK

If you have any suggestions or questions about `partpy` feel free to email me at nekroze@eturnilnetwork.com.

You can check out more of what I am doing at http://nekroze.eturnilnetwork.com my blog.

If you encounter any errors or problems with `partpy`, please let me know! Open an Issue at the GitHub http://github.com/Nekroze/partpy main repository.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX