pent Documentation

Release 0.2

Brian Skinn

Oct 28, 2019

CONTENTS

A common frustration in data analysis is software tooling that *only* generates its output in human-readable fashion. Thus, even if there is visible structure to the data, that structure is embedded in a format that can be awkward to parse.

Take the following toy data:

```
>>> text = """{lots of content}
...
...           $data1
...           0     0.000
...           1    -3.853
...           2     1.219
...
...           $data2
...           0     3.142
...           1     2.718
...           2     6.022
...
...           {lots more content}"""
```

Say it's needed to extract the list of decimal values in $data1, *without* the accompanying integers. Further, say that in any given particular output file, this list of values can be of *any* length.

One could write a line-by-line search to parse out the values, but that's a slow way to go about it if there are many such data blocks that need to be extracted.

Regex is a pretty natural tool to use here, but writing the regex to retrieve these values is a non-trivial task: because of the way regex capture groups work, you really have to write *two* regexes. The first regex captures the whole chunk of text of interest, and the second searches within that chunk to capture the values from the individual lines.

`pent` **writes all this regex for you.**

All you have to do is provide `pent` with the structure of the text using its custom mini-language, including which parts should be captured for output, and it will scrape the data directly from the text:

```
>>> prs = pent.Parser(
...     head="@.$data1",
...     body="#.+i #!..d",
... )
>>> prs.capture_body(text)
[[['0.000'], ['-3.853'], ['1.219']]]
```

This is just one example of `pent`'s parsing capabilities—it's an extremely flexible tool, which can retrieve just about anything you want from just about any surrounding text.

`pent` is available on PyPI, and thus can be installed via "pip install pent".

Usage instructions for `pent` are provided in the *tutorial*, broken up into (1) an explanation of the *basics* of the syntax and (2) exposition of a number of (more-)realistic *examples*. For those so inclined, a formal grammar of the mini-language is also *provided*.

CHAPTER

ONE

WHAT PENT IS NOT

pent is **not** well suited for parsing data with an extensively nested or recursive structure, especially if that structure is defined by clear rules. Have JSON, XML, or YAML? There are other libraries specifically made for those formats, and you should use them. pent ultimately is just a fancy regex generator, and thus it carries the same functional constraints. If you build a *Parser* that is too complex, it *will* run until approximately the heat death of the universe!

Contents:

1.1 pent Parser Tutorial

There is almost always more than one way to construct a pent *Parser* to capture a given dataset. Sometimes, if the data format is complex or contains irrelevant content interspersed with the data of interest, significant pre- or post-processing may be required. As well, it's important to inspect your starting data carefully, often by loading it into a Python string, to be sure there aren't, say, a bunch of unprintable characters floating around and fouling the regex matches.

This tutorial starts by describing the basic structure of the semantic components of pent's parsing model: *tokens*, *patterns*, and *Parsers*. It then lays out some approaches to constructing *Parsers* for realistic datasets, with the goal of enabling new users to get quickly up to speed building their own *Parsers*.

For a formal description of the grammar of the tokens used herein, see the *pent Mini-Language Grammar*.

1.1.1 Basic Usage

pent searches text in a line-by-line fashion, where a line of text is delimited by the start/end of the string, and/or by newline(s).

Each line of text to be matched by pent is represented by a *pattern*, passed into a *Parser*. Each *pattern* is a string composed of zero or more whitespace-separated *tokens*, which define in a structured way what the overall *pattern* should match. Both *patterns* **and** *tokens* can also include *flags*, which modify the semantics of how they are processed.

At present, whitespace is hardcoded to include only spaces and tab characters (\t). Various options for user-configurable whitespace definition are planned (#26).

Basic Usage: Tokens

pent understands four kinds of tokens, which match varying types of content. One is an *'any' token*, which matches an arbitrary span of whitespace and/or non-whitespace content. The other three types are intended to match specific kinds of content within the line of text that are often, but not always, separated from surrounding content by whitespace.

All four kinds of tokens accept a *flag* that instructs the encapsulating `Parser` to capture the content matching the token for output. A subset of the tokens accepts a *flag* that alters how the `Parser` handles the presence or absence of whitespace following the content matching the token.

Additionally, two of the four token types accept *required* arguments, which specify with more precision the content that the token should match. These required arguments are explained in the respective sections below.

The 'Any' Token: ~

The 'any' token will match **anything**, including a completely blank line. It behaves essentially the same as ".*" in regex.

Currently, the 'any' token only accepts the *'capture' flag* (becoming "~!"). Addition of support for the *'space-after' flags* is planned (#78).

Note that any content matched by a capturing 'any' token will be split at whitespace in `Parser` output.

The 'Misc' Token: &

The 'misc' token matches any sequence of non-whitespace characters. Its uses are similar to the *'any' token*, except that its match is confined to a single whitespace-delimited piece of content. It is mainly intended for use on non-numerical data whose content is not constant, and thus the *'literal' token* cannot be used.

The 'misc' token has one required argument, indicating whether it should match exactly one piece of content (&.) or one-or-more pieces of content (&+). When matching one-or-more, the 'misc' token interleaves *required* whitespace between each reptition.

At this time, the functional difference between "~" and "&+" is minimal.

The 'misc' token accepts both the *capture flag* and the *space-after* modifier flags.

The 'Literal' Token: @

The 'literal' token matches an *exact* sequence of one or more whitespace-delimited characters, which is provided as a required argument in the token definition.

Similar to the *'misc' token*, the 'literal' token also has the quantity specifier as a required argument: either "@." for exactly one match or "@+" for one-or-more matches.

The argument for the string to be matched follows the quantity argument. Thus, to match the text foo exactly once a suitable token might be "@.foo".

In the situation where it's needed to match a literal string containing a space, the entire token can be enclosed in quotes: "'@.this has spaces'".

The 'literal' token differs from the *'misc'* and *'number'* tokens in that when the one-or-more argument is used, it **prohibits** whitespace between the repetitions. This allows, e.g., a long sequence of hyphens to be represented by a token like "@+-". Similarly, a long sequence of alternating hyphens and spaces could be represented by "'@+- '".

The 'literal' token accepts both the *capture flag* and the *space-after* modifier flags.

The 'Number' Token:

The 'number' token allows for selectively matching numbers of varying types in the text being parsed; in particular, matches can be constrained by sign (positive, negative, or either) or by format (integer, decimal, or scientific notation; or, combinations of these).

The 'number' token takes three required, single-character arguments:

1. Quantity: #. for exactly one, or #+ for one-or-more.
2. Sign: #[.+]+ for positive, #[.+]- for negative, or #[.+]. for either sign.
3. Number Format: #[.+][.-+]i for integer, #[.+][.-+]d for decimal, #[.+][.-+]s for scientific notation, #[.+][.-+]f for float (decimal or scinot) #[.+][.-+]g for general (integer or float).

The ability to specify different types of number formatting was implemented for this token because it is often the case that numbers printed in different formats have different semantic significance, and it's thus useful to be able to filter/capture based on that format. *This example* illustrates a simplified case of this.

As with the *'misc' token*, when matching in one-or-more quantity mode, the 'number' token interleaves *required* whitespace between each reptition.

The 'number' token accepts both the *capture flag* and the *space-after* modifier flags.

Token Flags

Currently, two types of flags can be passed to tokens: *capture flag* and the *space-after* modifier flags.

If both flags are used in a given token, the space-after modifier flag must **precede** the capture flag.

Capture Flag: `!`

In most cases, not all of the data in a block of text is of interest for downstream processing. Thus, `pent` provides the token-level 'capture' flag, "!", which marks the content of that token for inclusion in the output of `capture_body()` and `capture_struct()`. The 'capture' flag is an integral part of all of the *tutorial examples*.

Space-After Flags: `o` and `x`

With no space-after flag provided, all tokens *REQUIRE* the presence of trailing whitespace (or EOL) in order to match. This is because most content is anticipated to be whitespace-delineated, and thus this default leads to more concise `Parser` definitions.

However, there are situations where changing this behavior is useful for defining a well-targeted `Parser`, and some where changing it is necessary in order to compose a functional `Parser` at all.

As an example, take the following line of text:

```
The foo is in the foo.
```

The token "@.foo" would match the first occurrence of the word "foo", because it has whitespace after it, but it would *not* match the second occurrence, since it is immediately followed by a period.

In order to match both occurrences, the 'optional trailing whitespace flag', "o", could be added, leading to the token "@o.foo".

If it were desired only to match the second occurrence, the 'prohibited trailing whitespace flag', "x", could be added, yielding "@x.foo".

This tutorial example provides further illustration of the use of these flags in more-realistic situations.

Basic Usage: Patterns

A `pent` *pattern* is a series of *tokens* that represents **all** non-whitespace content on a given line of text. Each token (along with its arguments and flags) is delimited by whitespace.

A blank line—one that is empty, or contains only whitespace—can be matched with an empty pattern string:

```
>>> check_pattern(pattern="", text="")
MATCH

>>> check_pattern(pattern="", text="          ")
MATCH

>>> check_pattern(pattern="", text="     \t    ")
MATCH
```

If a line contains one piece of non-whitespace text, a single token will suffice to match the whole line:

```
>>> check_pattern(pattern="&.", text="foo")
MATCH

>>> check_pattern(pattern="&.", text="     foo")
MATCH

>>> check_pattern(pattern="#..i", text="-5")
MATCH

>>> check_pattern(pattern="#..i", text="    50000   ")
MATCH

>>> check_pattern(pattern="#..f", text="2")  # Wrong number type
NO MATCH

>>> check_pattern(pattern="#.-i", text="2")  # Wrong number sign
NO MATCH

>>> check_pattern(pattern="", text="42")  # Line is not blank
NO MATCH
```

If a line contains more than one piece of non-whitespace text, **all pieces** must be matched by a token in the pattern:

```
>>> check_pattern(pattern="&+", text="foo bar baz")  # One-or-more gets all three
MATCH

>>> check_pattern(pattern="&. &.", text="foo bar baz")  # Only 2/3 words matched
NO MATCH

>>> check_pattern(pattern="&. #..i", text="foo 42")
MATCH

>>> check_pattern(pattern="&+ #..i", text="foo bar baz 42")
MATCH

>>> check_pattern(pattern="#+.i", text="-2 -1 0 1 2")
MATCH

>>> check_pattern(pattern="#+.i", text="-2 -1 foo 1 2")  # 'foo' is not an int
NO MATCH
```

(continues on next page)

(continued from previous page)

```
>>> check_pattern(pattern="#+.i &. #+.i", text="-2 -1 foo 1 2")
MATCH
```

Be careful when using "~" and "&+", as they **may** match more aggressively than expected:

```
>>> check_pattern(pattern="~ #+.i", text="foo bar 42 34")
MATCH

>>> show_capture(pattern="~! #+.i", text="foo bar 42 34")
[[['foo', 'bar']]]

>>> check_pattern(pattern="&+ #+.i", text="foo bar 42 34")
MATCH

>>> show_capture(pattern="&!+ #+.i", text="foo bar 42 34")
[[['foo', 'bar', '42']]]

>>> check_pattern(pattern="&+ #+.i", text="foo 42 bar 34")
MATCH

>>> show_capture(pattern="&!+ #+.i", text="foo 42 bar 34")
[[['foo', '42', 'bar']]]
```

Punctuation will foul matches unless explicitly accounted for:

```
>>> check_pattern(pattern="#+.i", text="1 2 ---- 3 4")
NO MATCH

>>> check_pattern(pattern="#+.i &. #+.i", text="1 2 ---- 3 4")
MATCH
```

In situations where punctuation is directly adjacent to the content to be captured, the *space-after flags* must be used to modify `pent`'s expectations for whitespace:

```
>>> check_pattern(pattern="~ #..d @..", text="The value is 3.1415.")  # No space␣
↪between number and '.'
NO MATCH

>>> check_pattern(pattern="~ #x..d @..", text="The value is 3.1415.")
MATCH
```

In situations where some initial content will definitely appear on a line, but some additional trailing content *may or may not* appear at the end of the line, it's important to use one of the space-after modifier flags in order for `pent` to find a match when the trailing content is absent. This is because the default required trailing whitespace will (naturally) *require* whitespace to be present between the end of the matched content and the end of the line, and if EOL immediately follows the content the pattern match will fail, since the required whitespace is absent:

```
>>> check_pattern(pattern="&. #.+i ~", text="always 42 sometimes")
MATCH

>>> check_pattern(pattern="&. #.+i ~", text="always 42")
NO MATCH

>>> check_pattern(pattern="&. #.+i ~", text="always 42   ")
MATCH
```

(continues on next page)

(continued from previous page)

```
>>> check_pattern(pattern="&. #x.+i ~", text="always 42")
MATCH

>>> check_pattern(pattern="&. #x.+i ~", text="always 42 sometimes")
MATCH
```

Optional Line Flag: ?

In some cases, an entire line of text will be present in some occurrences of a desired *Parser* match with a block of text, but absent in others. To accommodate such situations, pent recognizes an 'optional-line flag' in a pattern. This flag is a sole "?", occurring as the first "token" in the pattern. Inclusion of this flag will cause the pattern to match in the following three cases:

1. A line is present that completely matches the optional pattern (per usual behavior).
2. A blank line (no non-whitespace content) is present where the optional pattern would match.
3. **NO** line is present where the optional pattern would match.

It is difficult to construct meaningful examples of this behavior without using a full *Parser* construction; as such, see *this tutorial page* for more details.

Basic Usage: Parsers

The *Parser* is the main user-facing interface to pent, where the patterns matching the data of interest are defined. *Parsers* are created with three arguments, *head*, *body*, and *tail*. All *Parsers* must have a *body*; *head* and *tail* are optional.

A section of text matched by a given *Parser* will have the following structure:

- If *head* is defined, it will be matched exactly once, and its content must immediately precede the *body* content.
- *body* will be matched one or more times.
- If *tail* is defined, it will be matched exactly once, and its content must immediately follow the *body* content.

Each of *head*, *body*, and *tail* can be one of three things:

1. A single pent *pattern*, matching a single line of text
2. An ordered iterable (tuple, list, etc.) of patterns, matching a number of lines of text equal to the length of the iterable
3. A *Parser*, matching its entire contents

The syntax and matching structure of *Parsers* using these three kinds of arguments are illustrated below using trivial examples. Application of pent to more-realistic situations is demonstrated in the *Examples section* of the tutorial.

In the below examples, most illustrations are of the use of *head*, rather than *tail*. However, the principles apply equally well to both.

Matching with Single Patterns

The simplest possible *Parser* only has *body* defined, containing a single pent *pattern*:

```
>>> prs = pent.Parser(body="@!.bar")
>>> text = """foo
...           bar
...           baz"""
>>> prs.capture_body(text)
[[['bar']]]
```

As noted, *body* will match multiple times in a row:

```
>>> text = """foo
...           bar
...           bar
...           bar
...           baz"""
>>> prs.capture_body(text)
[[['bar'], ['bar'], ['bar']]]
```

Multiple occurrences of *body* in the text will match independently:

```
>>> text = """foo
...           bar
...           baz
...           bar
...           baz"""
>>> prs.capture_body(text)
[[['bar']], [['bar']]]
```

If only that first bar is of interest, the *Parser* match can be constrained with a *head*:

```
>>> prs_head = pent.Parser(head="@.foo", body="@!.bar")
>>> prs_head.capture_body(text)
[[['bar']]]
```

Adding just a *tail* doesn't really help, since baz follows both instances of bar:

```
>>> prs_tail = pent.Parser(body="@!.bar", tail="@.baz")
>>> prs_tail.capture_body(text)
[[['bar']], [['bar']]]
```

Matching with Iterables of Patterns

Sometimes data is structured in such a way that it's necessary to associate more than one line of text with a given portion of a *Parser*. This is most common with *head* and *tail*, but it can occur with *body* as well. These situations are addressed by using iterables of patterns when instantiating a *Parser*.

The following is a situation where the header portion of the data contains two lines, one being a string label and the other being a series of integers, and it's important to capture only the "wanted" data block:

```
>>> text = """WANTED_DATA
...           1    2     3
...           1.5  2.1   1.1
...
...           UNWANTED_DATA
...           1    2     3
...           0.1  0.4   0.2
```

(continues on next page)

(continued from previous page)

```
...             """
>>> pent.Parser(
...     head=("@.WANTED_DATA", "#++i"),
...     body="#!++d"
... ).capture_body(text)
[[['1.5', '2.1', '1.1']]]
```

Note that even though WANTED_DATA appears in the header line of the 'unwanted' data block, since the @.WANTED_DATA token does not match the *complete* contents of UNWANTED_DATA, the `Parser` does not match that second block.

If *head* were left out, or defined just to match the rows of integers, both datasets would be retrieved:

```
>>> pent.Parser(head="#++i", body="#!++d").capture_body(text)
[[['1.5', '2.1', '1.1']], [['0.1', '0.4', '0.2']]]
```

Situations calling for passing an iterable into *body* are less common, but can occur if there is a strictly repeating, cyclic pattern to the text to be parsed:

```
>>> text_good = """DATA
...                 foo
...                 bar
...                 foo
...                 bar
...                 foo
...                 bar"""
>>> prs = pent.Parser(
...     head="@.DATA",
...     body=("@!.foo", "@!.bar")
... )
>>> prs.capture_body(text_good)
[[['foo', 'bar'], ['foo', 'bar'], ['foo', 'bar']]]
```

Note in the .capture_body() output that even though each foo and bar appear on separate lines in the text, because the capture of each pair is defined as the *body* of a single `Parser`, they end up being treated as though they had been on the same line. Another example of this behavior can be found in *this tutorial example*.

If the lines of body text are not strictly cyclic-repeating, this approach won't work:

```
>>> text_bad = """DATA
...                foo
...                bar
...
...                foo
...                bar"""
>>> prs.capture_body(text_bad)
[[['foo', 'bar']]]
```

There are other approaches that can handle such situations, such as the *optional-line pattern flag*:

```
>>> pent.Parser(
...     head="? @.DATA",
...     body=("@!.foo", "@!.bar")
... ).capture_body(text_bad)
[[['foo', 'bar']], [['foo', 'bar']]]
```

Matching with a Nested `Parser`

For data with more complex internal structure, often the best way to match it is to pass a *Parser* to one or more of *head*, *body*, or *tail*.

In situations where the header or footer content has a variable number of lines that all match the same pattern, passing a *Parser* is often the most concise approach, as it exploits the implicit matching of one-or-more lines by the *body* of that internal *Parser*:

```
>>> text_head = """foo
...                 1 2 3
...                 bar
...                 bar
...
...                 foo
...                 1 2 3
...                 4 5 6 7 8
...                 9 10
...                 bar
...                 bar
...                 bar"""
>>> prs_head = pent.Parser(
...     head=pent.Parser(
...         head="@.foo",
...         body="#++i",
...     ),
...     body="@!.bar",
... )
>>> prs_head.capture_body(text_head)
[[['bar'], ['bar']], [['bar'], ['bar'], ['bar']]]
```

Another common use of an internal *Parser* is when the main data content itself has a header/body/footer structure, but it is also necessary to specify an overall header for the data in order to avoid capturing multiple times within the broader text:

```
>>> text_body = """WANTED
...                 foo
...                 bar
...                 bar
...
...                 UNWANTED
...                 foo
...                 bar
...                 bar
...                 bar
...                 bar"""
>>> prs_body = pent.Parser(
...     head="@.WANTED",
...     body=pent.Parser(
...         head="@.foo",
...         body="@!.bar",
...     ),
... )
>>> prs_body.capture_body(text_body)
[[['bar'], ['bar']]]
```

A clearer description of this approach is provided in *this tutorial example*.

1.1.2 Examples

This section of the tutorial contains examples of applications of `pent` to parsing of "real" (or, at least "real-like") datasets.

Capturing with a Single `Parser`

This first example is a modified version of the dataset used in the first half of the project README, drawn from a .hess file generated by ORCA:

```
>>> text = dedent("""\
... $vibrational_frequencies
... 6
... 0           0.000000
... 1           0.000000
... 2        -194.490162
... 3        -198.587114
... 4         389.931897
... 5         402.713910
... """)
```

A Minimal `Parser` Body

Focusing first on the main section of the data, the goal here is to retrieve the floats in the right-hand column; the rest of the content is irrelevant. However, the integers in the left-hand column still have to be represented in the pattern, even if they're not captured.

So, to represent those leading integers, the first token of the body pattern needs to be a single number (#.) that's not captured (omit !), with a positive sign (+) and integer format (i), leading to #.+i.

Then, to match the second, decimal value on each line, the second token needs to also be a single number (#.) of decimal format (d). But, since we want these values to be captured in output, it's necessary to insert ! after #. And, since some of the values in this list are negative and some are positive, the token should allow any sign (.). Thus, the second token should be #!..d.

So, a first stab at the body of the *`Parser`* would be:

```
>>> prs = pent.Parser(body="#.+i #!..d")
>>> prs.capture_body(text)
[[['0.000000'], ['0.000000'], ['-194.490162'], ['-198.587114'], ['389.931897'], ['402.
↪713910']]]
```

Works nicely! There are two things to note about the data returned here, though:

First, all of the numerical values are returned as **strings**. `pent` tries to maximize flexibility by making no assumptions about what needs to be done with the data. Thus, some post-processing will always be required. For example, to get the captured values from *data* into a `numpy` array, one could do the following:

```
>>> arr = np.asarray(prs.capture_body(text), dtype=float).squeeze()
>>> print(arr)
[   0.          0.       -194.490162 -198.587114  389.931897  402.71391 ]
```

Second, the captured data is always returned as a nested series of lists. In situations like this one, where a single *`Parser`* is used, the nesting will be three levels deep. This is because each matching block of data is returned as a matrix (a list of lists), and each of these matrices is then in turn a member of the outermost list.

In this particular instance, since the *body* captures exactly one value per line of test parsed, the innermost lists are length-one. And, since there are six lines that match the *body* pattern, the matrix that is returned is of size 6x1 (a list containing six length-one lists).

This means that if there had been a gap in the data, the outermost list would have had length greater than one:

```
>>> text2 = dedent("""\
... 0       0.000000
... 1       0.000000
...
... 2    -194.490162
... 3    -198.587114
... """)
>>> prs.capture_body(text2)
[[['0.000000'], ['0.000000']], [['-194.490162'], ['-198.587114']]]
```

There are two blocks of data here, each with two rows of one value each, so the return value from *capture_body()* is a length-two list, where each item of that list represents a 2x1 matrix.

Capturing Multiple Values per Line

If one wanted to also capture the integer indices in each row, the only change needed would be to add the ! capturing flag to that first token:

```
>>> pent.Parser(body="#!.+i #!..d").capture_body(text2)
[[['0', '0.000000'], ['1', '0.000000']], [['2', '-194.490162'], ['3', '-198.587114']]]
```

Constraining the `Parser` Match with a *head*

However, what if there are other datasets in the file that have this same format, but that we don't want to capture:

```
>>> text3 = dedent("""\
... $vibrational_frequencies
... 6
... 0           0.000000
... 1           0.000000
... 2        -194.490162
... 3        -198.587114
... 4         389.931897
... 5         402.713910
...
... $unrelated_data
... 3
... 0           3.316
... 1          -4.311
... 2          12.120
... """)
```

The original *Parser* will grab both of these blocks of data:

```
>>> prs.capture_body(text3)
[[['0.000000'], ['0.000000'], ['-194.490162'], ['-198.587114'], ['389.931897'], ['402.
↪713910']], [['3.316'], ['-4.311'], ['12.120']]]
```

The *Parser* can be constrained to only the data we want by introducing a *head* pattern:

```
>>> prs2 = pent.Parser(
...     head=["@.$vibrational_frequencies", "#!.+i"],
...     body="#.+i #!..d"
... )
>>> prs2.capture_body(text3)
[[['0.000000'], ['0.000000'], ['-194.490162'], ['-198.587114'], ['389.931897'], ['402.
↪713910']]]
```

This use of *head* introduces two concepts: (1) the 'literal string' token, @, in combination with the "." quantity marker telling the *Parser* to match the literal string exactly once; and (2) the `pent` feature wherein a length-*n* ordered iterable of pattern strings (here, length-two) will match *n* lines from the data string. In this case, the first string in the tuple matches the "$vibrational_frequencies" marker in the first line of the header, and the second captures the single positive integer in the second line of the header.

Capturing in *head* and *tail* with `capture_struct()`

In the example immediately above, note that even though the "!" capturing flag is specified in the second element of the *head*, that captured value does not show up in the *capture_body()* output. Captures in *head* and *tail* must be retrieved using *capture_struct()*:

```
>>> prs2.capture_struct(text3)
[{<ParserField.Head: 'head'>: [['6']], <ParserField.Body: 'body'>: [['0.000000'], ['0.
↪000000'], ['-194.490162'], ['-198.587114'], ['389.931897'], ['402.713910']],
↪<ParserField.Tail: 'tail'>: None}]
>>> prs2.capture_struct(text3)[0][pent.ParserField.Head]
[['6']]
```

The return value from *capture_struct()* has length equal to the number of times the *Parser* matched within the text. Here, since the pattern only matched once, the return value is of length one.

As a convenience, the lists returned by *capture_struct()* are actually of type *ThruList*, a custom subclass of `list`, which will silently pass through indices/keys to their first argument if and only if they are of length one. Thus, the following would also work for *prs2* operating on *text3*:

```
>>> prs2.capture_struct(text3)[pent.ParserField.Head]
[['6']]
```

But, it would break for the original *prs*, where the overall pattern matched twice:

```
>>> prs.capture_struct(text3)
[{<ParserField.Head: 'head'>: None, <ParserField.Body: 'body'>: [['0.000000'], ['0.
↪000000'], ['-194.490162'], ['-198.587114'], ['389.931897'], ['402.713910']],
↪<ParserField.Tail: 'tail'>: None}, {<ParserField.Head: 'head'>: None, <ParserField.
↪Body: 'body'>: [['3.316'], ['-4.311'], ['12.120']], <ParserField.Tail: 'tail'>:␣
↪None}]
>>> prs.capture_struct(text3)[pent.ParserField.Head]
Traceback (most recent call last):
    ...
pent.errors.ThruListError: Invalid ThruList index: Numeric index required for len != 1
```

As a final note, consider the difference between the *head* and *tail* results for the below *Parser*, where *head* is defined but has no capturing tokens present (yields `[[]]`), but *tail* is not specified (yields `None`):

```
>>> pent.Parser(head="#.+i", body="#.+i #!..d").capture_struct(text)
[{<ParserField.Head: 'head'>: [[]], <ParserField.Body: 'body'>: [['0.000000'], ['0.
↪000000'], ['-194.490162'], ['-198.587114'], ['389.931897'], ['402.713910']],
↪<ParserField.Tail: 'tail'>: None}]
```

(continues on next page)

(continued from previous page)

Capturing with Nested `Parser`s

`pent` is also able to parse and capture higher-dimensional data stored as free text. Take the following data string:

```
>>> text = dedent("""\
... $hessian
... 4
...                     0          1
...       0       0.473532   0.004379
...       1       0.004785   0.028807
...       2       0.004785  -0.022335
...       3      -0.418007   0.008333
...                     2          3
...       0       0.004379  -0.416666
...       1      -0.022335   0.008067
...       2       0.028807   0.008067
...       3       0.008333   0.420926
... """)
```

text represents a 4x4 matrix, with the first two columns printed in one section, and the second two columns printed in a separate, following section. Each row and column is marked with its respective index. In order to import this data successfully, the *body* of the main *Parser* will have to be set to a different, inner *Parser*.

Defining the Inner `Parser`

Each section of data columns starts with a row containing only positive integers, which does not need to be captured. After that leading row are multiple rows with data, each of which leads with a single positive integer, followed by decimal-format data of any sign:

```
>>> text_inner = dedent("""\
...                     0          1
...       0       0.473532   0.004379
...       1       0.004785   0.028807
...       2       0.004785  -0.022335
...       3      -0.418007   0.008333
... """)
```

One way to construct a *Parser* for this internal block is as follows:

```
>>> prs_inner = pent.Parser(
...     head="#++i",
...     body="#.+i #!+.d",
... )
>>> prs_inner.capture_body(text_inner)
[[['0.473532', '0.004379'], ['0.004785', '0.028807'], ['0.004785', '-0.022335'], ['-0.
↪418007', '0.008333']]]
```

Note that even though the multiple decimal values in each row of the data block were matched by the single "#!+.d" token in *body*, they were reported as separate values in the output. As currently implemented, `pent` will **always** split captured content at any internal whitespace; a further example of this with the 'any' token can be seen *here*.

Defining the Outer `Parser`

The outer *Parser* then makes use of the inner *Parser* as its *body*, with the two header lines defined in *head*:

```
>>> prs_outer = pent.Parser(
...     head=("@.$hessian", "#.+i"),
...     body=prs_inner,
... )
>>> data = prs_outer.capture_body(text)
>>> data
[[[['0.473532', '0.004379'], ['0.004785', '0.028807'], ['0.004785', '-0.022335'], ['-
↪0.418007', '0.008333']], [['0.004379', '-0.416666'], ['-0.022335', '0.008067'], ['0.
↪028807', '0.008067'], ['0.008333', '0.420926']]]]
```

Structure of the Returned *data*

The structure of the list returned by *capture_body()* nests four levels deep:

```
>>> arr = np.asarray(data, dtype=float)
>>> arr.shape
(1, 2, 4, 2)
```

This is because:

1. Each block of data is returned as a matrix (adds two levels);
2. The *body* of *prs_outer* is a *Parser* (adds one level); and
3. The *capture_body()* method wraps everything in a list (adds one level).

So, working from left to right, the (1, 2, 4, 2) shape of the data arises because:

1. The overall *prs_outer* matched **1 time**;
2. The inner *prs_inner*, as the *body* of *prs_outer*, matched **2 times**; and
3. Both blocks of data matched by *prs_inner* have **4 rows** and **2 columns**

Reassembling the Full 4x4 Matrix

In cases like this, `numpy`'s `column_stack()` provides a simple way to reassemble the full 4x4 matrix of data, though it is necessary to convert each matrix to an `ndarray` separately:

```
>>> np.column_stack([np.asarray(block, dtype=float) for block in data[0]])
array([[ 0.473532,  0.004379,  0.004379, -0.416666],
       [ 0.004785,  0.028807, -0.022335,  0.008067],
       [ 0.004785, -0.022335,  0.028807,  0.008067],
       [-0.418007,  0.008333,  0.008333,  0.420926]])
```

data[0] is used instead of *data* in the generator expression so that the two inner 4x2 blocks of data are yielded separately to `asarray()`.

Coping with Mismatched Data Block Sizes

Nothing guarantees that the data in a chunk of text will have properly matched internal dimensions, however. `pent` will still import the data, but it may not be possible to pull it directly into a `numpy` array as was done above:

```
>>> text2 = dedent("""\
... $hessian
... 4
...                     0          1
...       0       0.473532   0.004379
...       1       0.004785   0.028807
...       2       0.004785  -0.022335
...       3      -0.418007   0.008333
...                     2          3
...       0       0.004379  -0.416666
...       1      -0.022335   0.008067
... """)
>>> data2 = prs_outer.capture_body(text2)
>>> data2
[[['0.473532', '0.004379'], ['0.004785', '0.028807'], ['0.004785', '-0.022335'], ['-
↪0.418007', '0.008333']], [['0.004379', '-0.416666'], ['-0.022335', '0.008067']]]
>>> np.asarray(data2, dtype=float)
Traceback (most recent call last):
...
ValueError: setting an array element with a sequence.
>>> np.column_stack([np.asarray(block, dtype=float) for block in data2[0]])
Traceback (most recent call last):
...
ValueError: all the input array dimensions except for the concatenation axis must␣
↪match exactly
```

In situations like this, the returned data structure either must be processed with methods that can accommodate the missing data, or the missing data must be explicitly filled in before conversion to `ndarray`.

The Misc Token

Sometimes, data is laid out in text in a fashion where it cannot be matched using only numerical values. Either some elements of the data of interest are themselves non-numeric, or there are non-numeric portions of content interspersed with the numeric data of interest. `pent` provides the "misc" token (&) to handle these kinds of situations.

Take the following data, which is an example of the XYZ format for representing the atomic coordinates of a chemical system:

```
>>> text_xyz = dedent("""
... 5
... Coordinates from MeCl2F_2
...   C      -3.081564      2.283942      0.044943
...   Cl     -1.303141      2.255173      0.064645
...   Cl     -3.706406      3.411601     -1.180577
...   F      -3.541771      2.647036      1.270358
...   H      -3.439068      1.277858     -0.199370
... """)
```

In this case, pretty much everything in the text block is of interest. The first number indicates how many atoms are present (useful for cross-checking the data import), the line of text is an arbitrary string describing the chemical system, and the data block provides the atomic symbol of each atom and its xyz position in space.

The following *`Parser`* will enable capture of the entire contents of the string:

```
>>> prs_xyz = pent.Parser(
...     head=("#!..i", "~!"),
```

(continues on next page)

(continued from previous page)

```
...     body="&!. #!+.d",
... )
```

The atomic symbols and coordinates are most easily retrieved with *capture_body()*:

```
>>> data_atoms = prs_xyz.capture_body(text_xyz)
>>> data_atoms
[[['C', '-3.081564', '2.283942', '0.044943'], ['Cl', '-1.303141', '2.255173', '0.
↪064645'], ['Cl', '-3.706406', '3.411601', '-1.180577'], ['F', '-3.541771', '2.647036
↪', '1.270358'], ['H', '-3.439068', '1.277858', '-0.199370']]]
```

The atom count and description can be retrieved with *capture_struct()*:

```
>>> data_struct = prs_xyz.capture_struct(text_xyz)
>>> data_struct[pent.ParserField.Head][0]
['5', 'Coordinates', 'from', 'MeCl2F_2']
```

Unlike in *body*, where two-dimensional structure is inferred in captured data, in *head* and *tail* all captures are returned as elements of a single, flat `list`.

Currently, it is not possible to avoid the splitting of *all* captured content at whitespace, even it it was captured from a single 'any' or 'literal' token. #26 and/or #62 are planned and will provide mechanism(s) to change this behavior.

As an aside, in this particular case the 'misc' token was not strictly necessary in the *body*, as the capturing 'any' token (~!) would also have worked:

```
>>> prs_any = pent.Parser(
...     head=("#.+i", "~"),
...     body="~! #!+.d",
... )
>>> prs_any.capture_body(text_xyz)
[[['C', '-3.081564', '2.283942', '0.044943'], ['Cl', '-1.303141', '2.255173', '0.
↪064645'], ['Cl', '-3.706406', '3.411601', '-1.180577'], ['F', '-3.541771', '2.647036
↪', '1.270358'], ['H', '-3.439068', '1.277858', '-0.199370']]]
```

However, there are situations where the ability of the 'misc' token to match only a single, arbitrary piece of whitespace-delimited content is useful in order to narrow the specificity of the *Parser* match.

Another example of the use of the 'misc' token is given at **Post-Processing of Captured Data*.

*Post-Processing of Captured Data

Sometimes, data in text is laid out in a way such that `pent` can't retrieve *only* the data of interest using a *Parser*. In these cases, post-processing of the data obtained from *capture_body()* is the simplest approach.

Multiwfn LI

*Internal Spaces in One-Or-More Matches

Illustration of how misc/number and literal token types handle them differently.

The Optional-Line Token

In some situations, data is output in a fashion such that a line of, e.g., header text is present in some parts of the content of interest, but not others. Take the following fictitious example:

```
>>> text = dedent("""
...     $DATA
...     ITERATION 1
...       0     1     2
...       1.5   3.1   2.4
...       3     4     5
...       -0.1  2.7   -9.3
...     ITERATION 2
...       0     1     2
...       1.6   2.9   1.8
...       3     4     5
...       -0.4  2.1   -8.7
... """)
```

This data block could be matched with triply nested *Parsers*:

```
>>> prs_3x = pent.Parser(
...     head="@.$DATA",
...     body=pent.Parser(
...         head="@.ITERATION #..i",
...         body=pent.Parser(
...             head="#++i",
...             body="#!+.d",
...         ),
...     ),
... )
>>> prs_3x.capture_body(text)
[[[[['1.5', '3.1', '2.4']], [['-0.1', '2.7', '-9.3']]], [[['1.6', '2.9', '1.8']], [['-
↪0.4', '2.1', '-8.7']]]]]
```

However, that definition is quite bulky, and for more complex patterns and larger text inputs the three layers of nesting can sometimes lead to problematically slow parsing times.

The *optional-line* pattern flag allows for a simpler *Parser* structure here:

```
>>> prs_opt = pent.Parser(
...     head=("? @.$DATA", "@.ITERATION #..i"),
...     body=pent.Parser(
...         head="#++i",
...         body="#!+.d",
...     ),
... )
>>> prs_opt.capture_body(text)
[[[['1.5', '3.1', '2.4']], [['-0.1', '2.7', '-9.3']]], [[['1.6', '2.9', '1.8']], [['-
↪0.4', '2.1', '-8.7']]]]
```

The $DATA is now wrapped into the *head* of the outer of just *two* *Parsers*, flagged as optional so that the ITERATION 2 can be matched. This approach also returns the data with one fewer level of `list` enclosure, which may be convenient in downstream processing.

Since in this example the lines containing integers and the lines containing decimals are *strictly* alternating, yet another alternative would be to include the integer 'header' lines as a non-captured portion of the *body*:

```
>>> prs_opt = pent.Parser(
...     head=("? @.$DATA", "@.ITERATION #..i"),
...     body=pent.Parser(
...         body=("#++i", "#!+.d"),
...     )
... )
>>> prs_opt.capture_body(text)
[[[['1.5', '3.1', '2.4'], ['-0.1', '2.7', '-9.3']]], [[['1.6', '2.9', '1.8'], ['-0.4',
↪ '2.1', '-8.7']]]]
```

Doing it this way results in each ITERATION's data being grouped into a two-dimensional matrix, instead of each individual line of decimal values occurring in its own matrix. This may or may not be desirable, depending on the semantics of the data being captured.

The Three Cases of Optional-Line Matches

More generally, as noted at the *'pattern' basic usage page*, a pattern with the optional flag will match in three situations:

1. When a line is present matching the optional pattern:

```
>>> prs = pent.Parser(body=("@!.a", "? @!.b", "@!.c"))
>>> prs.capture_body("""a
...                     b
...                     c""")
[[['a', 'b', 'c']]]
```

2. When a blank line is present where the optional pattern would match:

```
>>> prs.capture_body("""a
...
...                     c""")
[[['a', None, 'c']]]
```

3. When there is **no** line present where the optional pattern would match:

```
>>> prs.capture_body("""a
...                     c""")
[[['a', None, 'c']]]
```

If a line is present that does not match the optional pattern, the **entire** *Parser* will fail to match:

```
>>> prs.capture_body("""a
...                     foo
...                     c""")
[]
```

Required/Optional/Prohibited Trailing Whitespace

By default, *number* (#), *misc* (&), and *literal* (@) tokens require trailing whitespace to be present in the text in order to match:

```
>>> text_space = dedent("""\
... foo: 5
... bar: 8
```

(continues on next page)

(continued from previous page)

```
... """)
>>> text_nospace = dedent("""\
... foo:5
... bar:8
... """)
>>> prs_req = pent.Parser(body="&. #!.+i")
>>> prs_req.capture_body(text_space)
[[['5'], ['8']]]
>>> prs_req.capture_body(text_nospace)
[]
```

`pent` provides a means to make this trailing whitespace either optional or prohibited, if needed, via a *token-level flag*.

Optional trailing whitespace is indicated with an "o" flag in the token:

```
>>> prs_opt = pent.Parser(body="&o. #!.+i")
>>> prs_opt.capture_body(text_space)
[[['5'], ['8']]]
>>> prs_opt.capture_body(text_nospace)
[[['5'], ['8']]]
```

Similarly, prohibited trailing whitespace is indicated with an "x" flag in the token:

```
>>> prs_prohib = pent.Parser(body="&x. #!.+i")
>>> prs_prohib.capture_body(text_space)
[]
>>> prs_prohib.capture_body(text_nospace)
[[['5'], ['8']]]
```

If used in combination with the capturing "!" flag, the trailing-space flag is placed *before* the capturing flag; e.g., as "&x!.".

One common situation where this capability is needed is when a number of interest is contained in prose text and falls at the end of a sentence:

```
>>> text_prose = dedent("""\
... pi is approximately 3.14159.
... """)
>>> pent.Parser(body="~ #!..d &.").capture_body(text_prose)
[]
>>> pent.Parser(body="~ #x!..d &.").capture_body(text_prose)
[[['3.14159']]]
```

Don't forget to include a token for that trailing period! The *Parser* won't find a match, otherwise:

```
>>> pent.Parser(body="~ #x!..d").capture_body(text_prose)
[]
```

Limitations of the "Any" Token

Note that, as currently implemented, the 'any' token (~) does not allow specification of optional or prohibited trailing whitespace; any content that it matches *must* be followed by whitespace for the *Parser* to work:

```
>>> text_sandwich = dedent("""\
... This number3.14159is sandwiched in text.
```

(continues on next page)

(continued from previous page)

```
... """)
>>> pent.Parser(body="~ #x!..d ~").capture_body(text_sandwich)
[]
```

In order to match this value, the preceding text must be matched either by a literal or a misc token:

```
>>> pent.Parser(body="~ @x.number #x!..d ~").capture_body(text_sandwich)
[[['3.14159']]]
>>> pent.Parser(body="~ &x. #x!..d ~").capture_body(text_sandwich)
[[['3.14159']]]
```

This deficiency will be addressed in #78.

*'Any' Tokens at EOL

TODO per #45

*Pre-Processing/Data Cleanup Example

pending

*Examples of `Parser`-Generated Regex

pending

** Incomplete*

1.2 pent Mini-Language Grammar

As discussed *here*, a `pent Parser` is constructed by passing it *patterns* composed of *tokens*. The grammar below specifies what constitutes a valid `pent` *token*.

For completeness, even though the *optional-line pattern flag* is called a *flag* and not a *token*, internally `pent` parses this flag as though it were a token, and thus it is included here.

This grammar is expressed in an approximation of extended Backus-Naur form. Content in double quotes represents a literal string, the pipe character indicates alternatives, square brackets indicate *optional* token flags, and parentheses indicate *required* token flags.

Grammar

```
token                   ::=  optional_line_flag | content_token

optional_line_flag      ::=  "?"
content_token           ::=  any_token | literal_token | misc_token | number_token

any_token               ::=  "~"[capture]
literal_token           ::=  "@"[space_after][capture](quantity)(literal_content)
misc_token              ::=  "&"[space_after][capture](quantity)
number_token            ::=  "#"[space_after][capture](quantity)(sign)(num_type)

space_after             ::=  optional_space_after | no_space_after
```

(continues on next page)

(continued from previous page)

```
optional_space_after    ::=  "o"
no_space_after          ::=  "x"

capture                 ::=  "!"

quantity                ::=  match_one | match_one_or_more
match_one               ::=  "."
match_one_or_more       ::=  "+"

sign                    ::=  any_sign | positive_sign | negative_sign
any_sign                ::=  "."
positive_sign           ::=  "+"
negative_sign           ::=  "-"

num_type                ::=  integer | decimal | sci_not | float | general
integer                 ::=  "i"
decimal                 ::=  "d"
sci_not                 ::=  "s"
float                   ::=  "f"
general                 ::=  "g"
```

1.3 API (draft page)

Unstructured API dump, to provide cross-reference targets for other portions of the docs.

Any of the objects/attributes/methods documented here may become private implementation details in future versions of `pent`.

Mini-language parser for `pent`.

`pent` Extracts Numerical Text.

Author Brian Skinn (bskinn@alum.mit.edu)

File Created 8 Sep 2018

Copyright (c) Brian Skinn 2018-2019

Source Repository http://www.github.com/bskinn/pent

Documentation http://pent.readthedocs.io

License The MIT License; see LICENSE.txt for full license terms

Members

class `pent.parser.`**Parser**(*head=None*, *body=None*, *tail=None*)

Mini-language parser for structured numerical data.

capture_body(*text*)

Capture all values from the pattern body, recursing if needed.

classmethod capture_parser(*prs*, *text*)

Perform capture of a Parser pattern.

classmethod capture_section(*sec*, *text*)

Perform capture of a str, iterable, or Parser section.

classmethod capture_str_pattern(*pat_str*, *text*)

Perform capture of string/iterable-of-str pattern.

capture_struct(*text*)
Perform capture of marked groups to nested dict(s).

classmethod convert_line(*line*, *, *capture_groups=True*, *group_id=0*)
Convert line of tokens to regex.

The constructed regex is required to match the entirety of a line of text, using lookbehind and lookahead at the start and end of the pattern, respectively.

group_id indicates the starting value of the index for any capture groups added.

classmethod convert_section(*sec*, *capture_groups=False*, *capture_sections=True*)
Convert the head, body or tail to regex.

static generate_captures(*m*)
Generate captures from a regex match.

pattern(*capture_sections=True*)
Return the regex pattern for the entire parser.

The individual capture groups are NEVER inserted when regex is generated this way.

Instead, head/body/tail capture groups are inserted, in order to subdivide matched text by these subsets. These 'section' capture groups are ONLY inserted for the top-level Parser, though – they are suppressed for inner nested Parsers.

Token handling for mini-language parser for `pent`.

`pent` Extracts Numerical Text.

Author Brian Skinn (bskinn@alum.mit.edu)

File Created 20 Sep 2018

Copyright (c) Brian Skinn 2018-2019

Source Repository http://www.github.com/bskinn/pent

Documentation http://pent.readthedocs.io

License The MIT License; see LICENSE.txt for full license terms

Members

class `pent.token.`**Token**(*token*, *do_capture=True*)
Encapsulates transforming mini-language patterns tokens into regex.

property capture
Return flag for whether a regex capture group should be created.

do_capture
Whether group capture should be added or not

property is_any
Return flag for whether the token is an "any content" token.

property is_misc
Return flag for whether the token is a misc token.

property is_num
Return flag for whether the token matches a number.

property is_optional_line
Return flag for whether the token flags an optional line.

property is_str
Return flag for whether the token matches a literal string.

property match_quantity
Return match quantity.

`None` for *pent.enums.Content.Any* or *pent.enums.Content.OptionalLine*

needs_group_id
Flag for whether group ID substitution needs to be done

property number
#: Return number format; `None` if token doesn't match a number.

property pattern
Return assembled regex pattern from the token, as `str`.

property sign
#: Return number sign; `None` if token doesn't match a number.

property space_after
Return Enum value for handling of post-match whitespace.

token
Mini-language token string to be parsed

Regex patterns for `pent`.

`pent` Extracts Numerical Text.

Author Brian Skinn (bskinn@alum.mit.edu)

File Created 2 Sep 2018

Copyright (c) Brian Skinn 2018-2019

Source Repository http://www.github.com/bskinn/pent

Documentation http://pent.readthedocs.io

License The MIT License; see LICENSE.txt for full license terms

Members

pent.patterns.**number_patterns = {(<Number.Decimal: 'd'>, <Sign.Positive: '+'>): '[+]?(**
`dict` of `pyparsing` patterns matching single numbers.

pent.patterns.**std_num_punct = 'deDE+.-'**
`str` with the standard numerical punctuation to include as not marking word boundaries. *de* is included to account for scientific notation.

pent.patterns.**std_scinot_markers = 'deDE'**
`str` with the standard allowed scientific notation exponent marker characters

pent.patterns.**std_word_chars = 'a-zA-Z0-9deDE+.-'**
Standard word marker characters for pent

pent.patterns.**std_wordify**(*p*)
Wrap a token in the `pent` standard word start/end markers.

pent.patterns.**std_wordify_close**(*p*)
Append the standard word end markers.

pent.patterns.**std_wordify_open**(*p*)
Prepend the standard word start markers.

pent.patterns.**wordify_close**(*p*, *word_chars*)
Append the word end markers.

pent.patterns.**wordify_open**(*p*, *word_chars*)
Prepend the word start markers.

pent.patterns.**wordify_pattern**(*p*, *word_chars*)
Wrap pattern with word start/end markers using arbitrary word chars.

Enums *for* pent.

pent Extracts Numerical Text.

Author Brian Skinn (bskinn@alum.mit.edu)

File Created 3 Sep 2018

Copyright (c) Brian Skinn 2018-2019

Source Repository http://www.github.com/bskinn/pent

Documentation http://pent.readthedocs.io

License The MIT License; see LICENSE.txt for full license terms

Members

class pent.enums.**Content**
Enumeration for the possible types of content.

Any = '~'
Arbitrary match, including whitespace

Misc = '&'
Arbitrary single-"word" match, no whitespace

Number = '#'
Number

OptionalLine = '?'
Flag to mark pattern line as optional

String = '@'
Literal string

class pent.enums.**Number**
Enumeration for the different kinds of recognized number primitives.

Decimal = 'd'
Decimal floating-point value; no scientific/exponential notation

Float = 'f'
"Floating-point value with or without an exponent

General = 'g'
"General" value; integer, float, or scientific notation

Integer = 'i'
Integer value; no decimal or scientific/exponential notation

SciNot = 's'
Scientific/exponential notation, where exponent is *required*

class pent.enums.**ParserField**
Enumeration for the fields/subsections of a Parser pattern.

Body = 'body'
Body

Head = 'head'
Header

Tail = 'tail'
Tail/footer

class pent.enums.**Quantity**
Enumeration for the various match quantities.

OneOrMore = '+'
One-or-more match

Single = '.'
Single value match

class pent.enums.**Sign**
Enumeration for the different kinds of recognized numerical signs.

Any = '.'
Any sign

Negative = '-'
Negative value only (leading '-' required; includes negative zero)

Positive = '+'
Positive value only (leading '+' optional; includes zero)

class pent.enums.**SpaceAfter**
Enumeration for the various constraints on space after tokens.

Optional = 'o'
Optional following space

Prohibited = 'x'
Following space prohibited

Required = ''
Default is required following space; no explicit enum value

class pent.enums.**TokenField**
Enumeration for fields within a mini-language number token.

Capture = 'capture'
Flag to ignore matched content when collecting into regex groups

Number = 'number'
Format of the numerical value (int, float, scinot, decimal, general)

Quantity = 'quantity'
Match quantity of the field (single value, optional, one-or-more, zero-or-more, etc.)

Sign = 'sign'
Sign of acceptable values (any, positive, negative)

SignNumber = 'sign_number'
Combined sign and number, for initial pattern group retrieval

SpaceAfter = 'space_after'
Flag to change the space-after behavior of a token

Str = 'str'
Literal content, for a string match

Type = 'type'
Content type (any, string, number)

Custom exceptions for `pent`.

`pent` Extracts Numerical Text.

Author Brian Skinn (bskinn@alum.mit.edu)

File Created 10 Sep 2018

Copyright (c) Brian Skinn 2018-2019

Source Repository http://www.github.com/bskinn/pent

Documentation http://pent.readthedocs.io

License The MIT License; see LICENSE.txt for full license terms

Members

exception `pent.errors.`**LineError**(*line*)
Raised during attempts to parse invalid token sequences.

exception `pent.errors.`**PentError**
Superclass for all custom pent errors.

exception `pent.errors.`**SectionError**(*msg=''*)
Raised from failed attempts to parse a Parser section.

exception `pent.errors.`**ThruListError**(*msg=''*)
Raised from failed ThruList indexing attempts.

exception `pent.errors.`**TokenError**(*token*)
Raised during attempts to parse an invalid token.

Custom list object for `pent`.

`pent` Extracts Numerical Text.

Author Brian Skinn (bskinn@alum.mit.edu)

File Created 3 Oct 2018

Copyright (c) Brian Skinn 2018-2019

Source Repository http://www.github.com/bskinn/pent

Documentation http://pent.readthedocs.io

License The MIT License; see LICENSE.txt for full license terms

Members

class `pent.thrulist.`**ThruList**
List that passes through *key* if len == 1.

Utility functions for `pent`.

`pent` Extracts Numerical Text.

Author Brian Skinn (bskinn@alum.mit.edu)

File Created 14 Oct 2018

Copyright (c) Brian Skinn 2018-2019

Source Repository http://www.github.com/bskinn/pent

Documentation http://pent.readthedocs.io

License The MIT License; see LICENSE.txt for full license terms

Members

pent.utils.**column_stack_2d**(*data*)
Perform column-stacking on a list of 2d data blocks.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Q

R

S

T

W