Published in Analytics Vidhya · Follow

Andrew Matteson · Follow

Feb 1, 2021 · 7 min read ★

Parsing PGN Chess Games With Python

I have a couple of projects in mind for analysis of chess game positions. They all require collecting a large number of chess games into structured data. This Week in Chess provides records of thousands of games each week. Each week consists of a file in Portable Game Notation (PGN), a plain text format for recording chess games.

I want to analyze hundreds of thousands of games, so step one is to gather the plain text files and parse them using the grammar of of PGN files.

If you just want the code to parse PGN, you can jump over to the GitHub repo. Otherwise, read on.

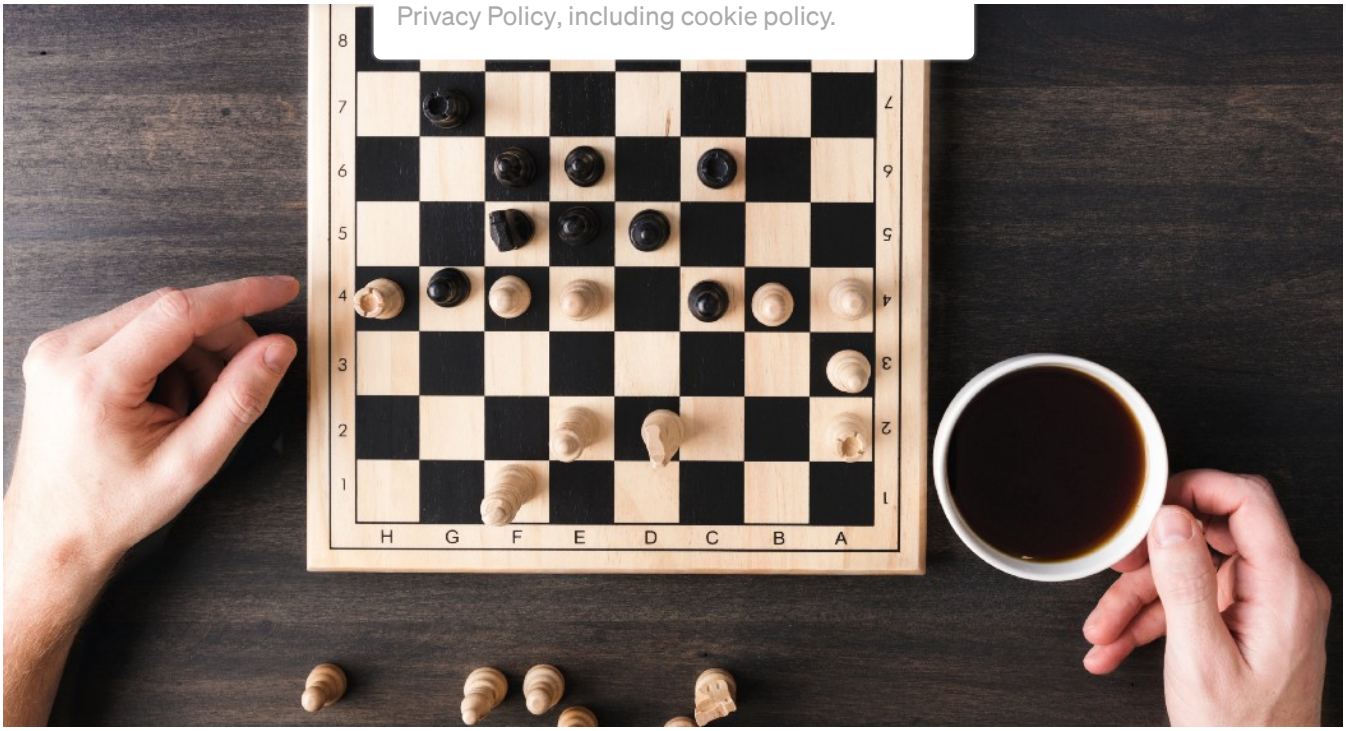


Photo by Sarah Pflug, used under CC0

The Grammar of PGN Files

Lots of text data, PGN included, obeys a “grammar”. The grammar of PGN includes annotations that are pairs of a tag and a string, the moves of the game, and the outcome. The important insight is that there are rules for how that text looks. For instance, this 2021 game between Magnus Carlsen and Jorden Van Foreest:

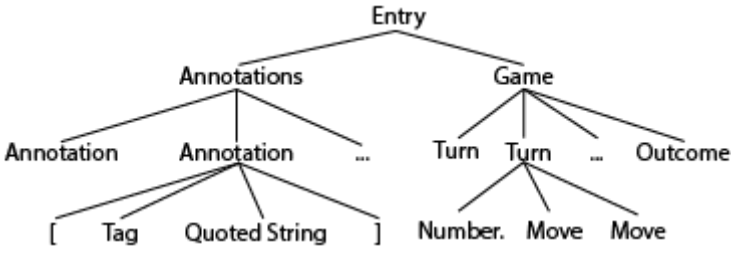
```
[Event "83rd Tata Steel Masters"]
[Site "Wijk aan Zee NED"]
[Date "2021.01.19"]
[Round "4.1"]
[White "Van Foreest,Jorden"]
[Black "Carlsen,M"]
[Result "1/2-1/2"]
[WhiteTitle "GM"]
[BlackTitle "GM"]
[WhiteElo "2671"]
[BlackElo "2862"]
[ECO "C78"]
[Opening "Ruy Lopez"]
[Variation "Archangelsk (counterthrust) variation"]
[WhiteFideId "1039784"]
[BlackFideId "1503014"]
[EventDate "2021.01.16"]
```

```
Rb6 24. Rxa4 Qxb2                                    4 Rf6 26. Rd7 Kg6
29. Kf1 Rxg4 30. R                                   . Kg2 Rd4 34. Ne2
Rxd3 35. Ng1 Rd7 36. Nf3 Kg7 37. h3 Bf6 38. Kg3 Rb7 39. Kg2 Re7 40.
Ra5 Rc7 41. Rd5 Ra7 42. Rb5 Be7 43. Nd4 Rd7 44. Nf3 Rd6 45. Rb7 Kf6
46. Ra7 h6 47. Nh4 Bd8 48. Rh7 Rd2 49. Rxh6 Kg7 50. Rxg6+ Kh7 51.
Nf3 Rxf2+ 52. Kxf2 Kxg6 53. Kg2 Kh5 54. Nd4 f4 55. Ne6 Bg5 56. Nxg5
Kxg5 57. Kf3 Kh4 58. Kxf4 Kxh3 1/2-1/2
```

The annotations appear in brackets, starting with the tag followed by a quoted string. After the annotations, the moves are listed by move number followed by white's move, then black's move (in algebraic notation). The last text declares the outcome, `1-0` for white wins, `0-1` for black, and `1/2-1/2` for a draw.

We're going to take a file of PGN entries and parse them. In computer science, a parser-combinator is a higher order function that takes in multiple parsers and returns a single parser. The topic is pretty deep, but the gist is that parser-combinators allow us to write out a parse tree of a grammar and iteratively build complex parsers by combining simpler ones. For this project, I use parsita, an open source python library for parsing written by one of my colleagues.

Writing out the parse tree from the file above is straightforward. An "entry" in the file is composed of two blocks: an "annotations" block and a "game" block. The annotations block is a new line delimited list of the sequence `[` , `TAG` , `"` , `STRING` , `"` , `]` . The game is a list of `NUMBER.` , `MOVE` , `MOVE` , followed by an `OUTCOME` . There's some trickiness about the last move because white might make one more move than black, but we'll deal with that later. It's best to visualize our grammar as a tree:



PGN Parse Tree (image by author)

Parsing Simple Text

We'll walk through each line of this code:

```
# Parse the " token
quote = lit(r'"')

# Parse a sequence of unicode characters excluding spaces
tag = reg(r'[\u0021-\u0021\u0023-\u005A\u005E-\u007E]+')

# parse a sequence of unicode characters including spaces
string = reg(r'[\u0020-\u0021\u0023-\u005A\u005E-\U0010FFFF]+')

# An annotation is combines a tag, and a quoted string delimited
# by other characters. We only care about the tag and string.
annotation = '[' >> tag << ' ' & (quote >> string << quote) << ']'

# Function to take a list of parsed annotations and convert to a
# dictionary
def formatannotations(annotations):
    return {ant[0]: ant[1] for ant in annotations}

# The annotation block is a list of 0 or more annotations.
annotations = repsep(annotation, '\n') > formatannotations
```

Parsita does the heavy lifting with literal parsers and regular expression parsers. The simplest parsing operation is to match a single character, the literal. I show two ways of doing this: `lit(r'"')` matches a quote; alternatively, parsita will accept a Python string into a parser and treat it as a literal. The brackets are parsed using the second technique.

The power of parsita comes from the five parsers we use in the above:

- `A & B`: sequential parser. Match the parser for `A` and the parser for `B`. Return both in a list.
- `A >> B`: discard left parser. Match the same as `&`. Return only `B`.
- `A << B`: discard right parser. Vice-versa from above.

- `repsep(A)` : repeated … return a list of matches.

So, expanding on the line defining the annotation:

```
annotation = '[' >> tag << ' ' & (quote >> string << quote) << ']'
```

An annotation is in sequence:

- a bracket (discard),
- a string without spaces,
- a space (discard),
- a quote (discard),
- a string,
- a quote (discard),
- a bracket (discard).

```
# Call the parser --> results in Success or Failure.
# .or_die() either gives the value of the success or raises an error

print(annotation.parse('[parsing "is cool"]').or_die())
> ['parsing', 'is cool']

# Annotations block converts the text to a dict
text = '''[parsing "is cool"]
[second "line"]'''

print(annotations.parse(text).or_die())
> {'parsing': 'is cool', 'second': 'line'}
```

Parsing More Complex Text

- `A|B` : the alternative ls, then try parser `B` .
- `opt(A)` : the optional parser. Tries to match parser `A` , if it succeeds return the value of `A` in a length one list. If it fails, return an empty list.

We use the alternative parser to simplify the definition of a move:

```
nullmove = lit('--') # Illegal move rarely used in annotations
longcastle = reg(r'O-O-O[+#]?')
castle = reg(r'O-O[+#]?')
regularmove = reg(r'[a-h1-8NBRQKx\+#=]+') # Matches more than just
chess moves
move = regularmove | longcastle | castle | nullmove
```

We match “regular” moves with a regular expression parser. Algebraic notation of a move references 1 or 2 squares with a-h, their file, and 1–8, their rank. Pieces have one letter designations. There are additional notations for capture (`x`), check (`+`), checkmate (`#`), and promotion (`=`). Other, non-regular, moves are long castle, castle, and null move (rarely encountered, but appears for various reasons).

So, a move tries to match a regular move, if it fails, tries to match long castle, then castle, then finally the null move.

```
def handleoptional(optionalmove):
    if len(optionalmove) > 0:
        return optionalmove[0]
    else:
        return None

def formatgame(game):
    return {
        'moves': game[0],
        'outcome': game[1]
    }

whitespace = lit(' ') | lit('\n')

movenumber = (reg(r'[0-9]+') << '.' << whitespace) > int
```

```
draw = lit('1/2-1/
white = lit('1-0')
black = lit('0-1')
outcome = draw | white | black

game = (rep(turn) & outcome) > formatgame
```

We need the optional parser because a game may terminate after white moves. When this occurs, in PGN, the file has a final move number and then a single move listed. So, the last “turn” of our grammar is optional. We parse this into a more useful form for Python by indexing into the returned optional list if it’s not empty, and otherwise returning `None`.

Putting it All Together

All that remains is to download a PGN file, and parse it using the code outlined above.

```
from parsita import *
from parsita.util import constant
import json

# Conversion functions
def formatannotations(annotations):
    return {ant[0]: ant[1] for ant in annotations}

def formatgame(game):
    return {
        'moves': game[0],
        'outcome': game[1]
    }

def formatentry(entry):
    return {'annotations': entry[0], 'game': entry[1]}

def handleoptional(optionalmove):
    if len(optionalmove) > 0:
        return optionalmove[0]
    else:
        return None

# PGN Grammar
quote = lit(r'"')
tag = reg(r'[\u0021-\u0021\u0023-\u005A\u005E-\u007E]+')
```



```
nullmove = lit('--') # Illegal move rarely used in annotations
longcastle = reg(r'O-O-O[+#]?')
castle = reg(r'O-O[+#]?')
regularmove = reg(r'[a-h1-8NBRQKx\+#=]+') # Matches more than just
chess moves
move = regularmove | longcastle | castle | nullmove

movenumber = (reg(r'[0-9]+') << '.' << whitespace) > int
turn = movenumber & (move << whitespace) & (opt(move << whitespace)
> handleoptional)

draw = lit('1/2-1/2')
white = lit('1-0')
black = lit('0-1')
outcome = draw | white | black

game = (rep(turn) & outcome) > formatgame

entry = ((annotations << rep(whitespace)) & (game <<
rep(whitespace))) > formatentry

file = rep(entry)

# Parse the file
with open('twic1368.pgn', 'r') as f:
    parsedoutput = file.parse(f.read()).or_die()
```

The parsed output is a list of games. Each game is structured in the same way:

```
{
  'annotations': {
    'Event': 'Lozovatsky Mem A 2021',
    'Site': 'Chelyabinsk RUS',
    'Date': '2021.01.18',
    'Round': '5.18',
    'White': 'Mischuk,D',
    'Black': 'Bryakin,M',
    'Result': '1-0',
    ...
  },
 'game': {
   'moves': [
      [1, 'd4', 'd5'],
      [2, 'c4', 'c6'],
      [3, 'Nc3', 'Nf6'],
```

Now, I'm off to find fun things to do with thousands of chess games.