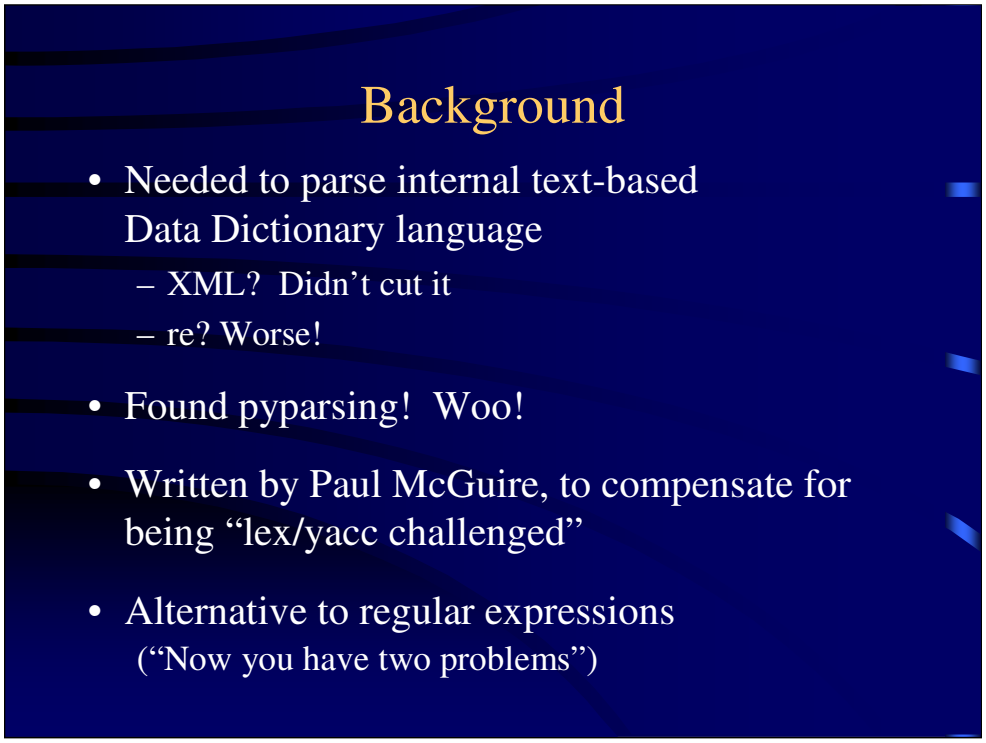
pyparsing

A User's Perspective

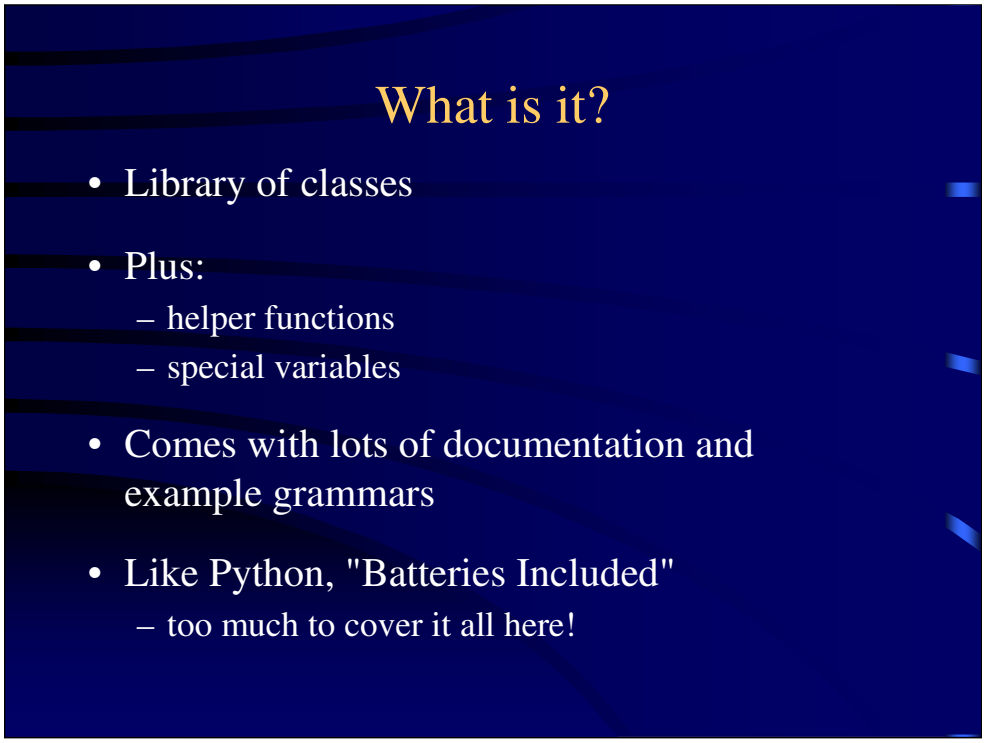
Wilson Fowlie, Delta Controls, Inc.
wfowlie@deltacontrols.com



My background with pyparsing:

Needed to parse internal Data Dictionary language.
XML didn't cut it. RegExes worse!

Searched online for a parser and found pyparsing.
Paul was very helpful when I ran into problems.



***The Zen of Pyparsing* (according to Paul McGuire)**

- Don't clutter up the parser grammar with white space, just handle it!
- Likewise for comments.
- Grammars must tolerate change, as grammar evolves or input text becomes more challenging.
- The Zen of Pyparsing
- Grammars do not have to be exhaustive to be useful.
- Simple grammars can return lists; complex grammars need named results.
- Class names are easier to read and understand than specialized typography.
- Parsers can (and sometimes should) do more than tokenize.

The Zen of Pyparsing Development

- Grammars should be:
- easy (and quick) to write
- easy to read
- easy to update
- No separate code-generation step.
- No imposed naming conventions.
- Stay Pure Python.

Paul McGuire:
ptmcg@users.sourceforge.net

Basic Building Block: Token Classes (ParserElement)

- Fixed-text Tokens:
  - Literal / CaselessLiteral

```
BDFL = Literal("Guido")
```

  - Keyword / CaselessKeyword

```
ifKwd = Keyword("if")
```

(Caseless versions always return original string)

Variable-text Token Classes

- Word

```
firstName = Word("ABC", "def")
```

  - if 2 strings, first matches first character only
- CharsNotIn (‘opposite’ of Word)

```
toPunc = CharsNotIn(",.;:!?")
```

  - Specify length with `min`, `max` or `exact` parameters

```
IPnum = Word("012346789", max=3)
```

Variable-text Token Classes

- Do you really miss `re`?
- Just gotta use a regular expression?
  - I doubt it …
  - … but if you absolutely must …
- Regex
  - initialize with an expression recognized by the Python `re` module.
- But really, look for a pyparsing solution first
  - … you'll be glad you did.

How to Use It (Basic Version)

- Assign variable to token or expression object
- Call `parseString()` or `parseFile()` method
  - returns a ParseResult object
    - more about that later; for now, think of it as a list

```
apples = Word("A", "elps")
apples.parseString("Appeels")
      ⇨ ['Appeels']
apples.parseString("Apfels")
      ⇨ ParseException
```

Expression Classes

- And
  - match all tokens in order
  - shortcut: + operator
- Or
  - match any token in list
  - matches whichever ‘uses up’ the most characters
  - shortcut: ^ operator

NOTE: operators will converts strings to Literals automatically!

Examples for And:

```
sentence = And([Literal("The"), Literal("quick"),
Literal("brown"), …])
sentence = Literal("The") + "quick" + "brown" + …
```

```
Examples for Or:
number = Or([integerNum, realNum])
number = integerNum ^ realNum
```

More Expression Classes

- MatchFirst
  - match any tokens in list
  - matches first one it comes to
    (as you might guess)
  - shortcut: | operator
- Each
  - must match all items in list, in any order
  - no shortcut operator
- ZeroOrMore, OneOrMore
  - Exactly what the names imply

Examples for MatchFirst:

```
loopStart = MatchFirst([whileKwd, doKwd])
loopStart = whileKwd | doKwd
```

Example for Each:

```
ident = Each([name, rank, serialNo])
ident.parseString("001 Guido BDFL")
      ⇨ ['001', 'Guido', 'BDFL']
```

Converter Classes

- Group
  - nested results
- Combine
  - concatenate results
- Suppress
  - leaves matches out of results
- Optional
  - no exception raised if no match

Example for Group:

```
ident = Group(Each([name, rank, serialNo]))
ident.parseString(“001 Guido BDFL”)  ⇨  [['001', 'Guido', 'BDFL']]
```

Example for Combine:

```
IPAdd = IPNum + '.' + IPNum + '.' + IPNum + '.' + IPNum
IPAdd.parseString("24.142.6.1")  ⇨  ['24', '.', ‘142', '.', '6', '.', '1']

IPAdd = Combine(IPAdd)
IPAdd.parseString("242.141.68.001")  ⇨  ["242.141.68.001"]
```

Example for Suppress:

```
dr = Suppress(Word('ABCDEFGHIJKLMNOPQRSTUVWXYZ', ':', exact = 2))
sep = Suppress(Word(r'\/', exact = 1))
part = sep + CharsNotIn(r'\/:*?"<>|')
path = dr + OneOrMore(part)

path.parseString("C:\Data\myData.txt")  ⇨  ['Data', 'myData.txt']
```

Helpful ParserElement Methods (besides `parse*()`)

- `scanString()`, `transformString()`, `searchString()`
- `copy()`
- `ignore()`
- `enablePackrat()`

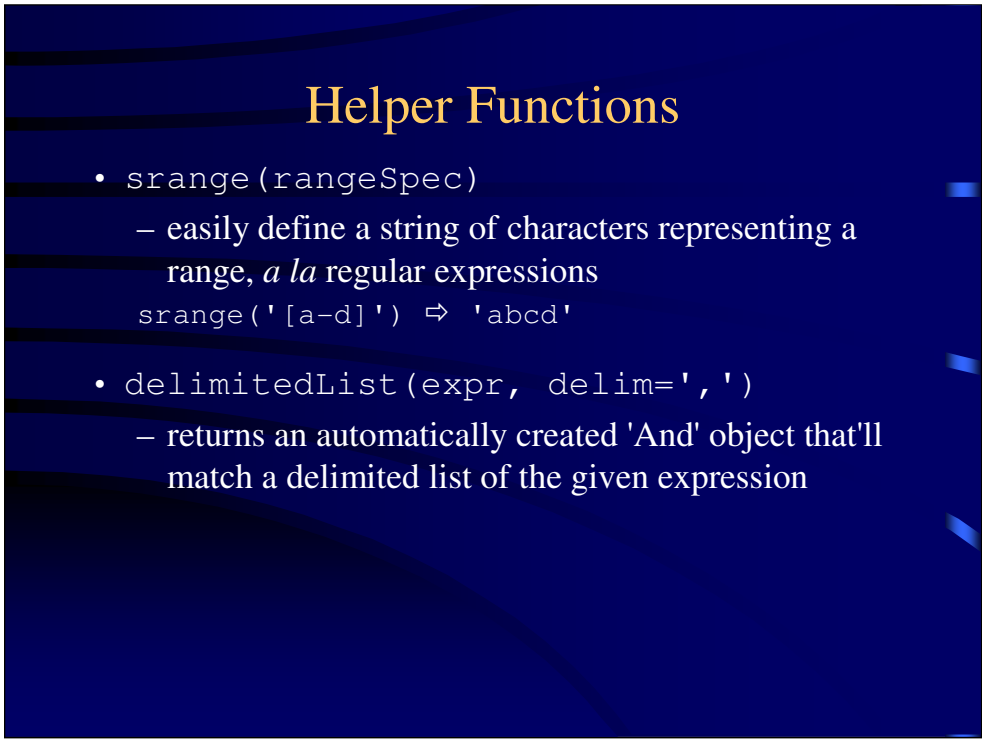
scanString(self, instring)
Scan the input string for expression matches.

transformString(self, instring)
Extension to scanString, to modify matching text with modified tokens that may be returned from a parse action.

searchString(self, instring)
Another extension to scanString, simplifying the access to the tokens found to match the given parse expression.

ignore(self, other)
Define expression to be ignored (e.g., comments) while doing pattern matching; may be called repeatedly, to define multiple comment or other ignorable patterns.

enablePackrat() *(Static method)*
Enables "packrat" parsing, which adds memoizing to the parsing logic.



The functions are in the pyparsing namespace/scope.

The srange function does NOT support any other regular expression syntax.

More Helper Functions

- Within-string locators:
  - line(), col(), lineno()
    - all use \n as line separator
- `oneOf(strs[, caseless[, useRegex]])`
  - strs ⇨ space-delimited literals
- setResultsName()
  - access nested parse results by name
  - more about this later

oneOf(strs, caseless=False, useRegex=True)

Helper to quickly define a set of alternative Literals, and makes sure to do longest-first testing when there is a conflict, regardless of the input order, but returns a MatchFirst for best performance.

•strs - a string of space-delimited literals, or a list of string literals

•caseless - (default=False) - treat all literals as caseless

•useRegex - (default=True) - as an optimization, will generate a Regex object; otherwise, will generate a MatchFirst object (if caseless=True, or if creating a Regex raises an exception)

setResultsName(self, name, listAllMatches)

Define name for referencing matching tokens as a nested attribute of the returned parse results.

More Power: Parse Actions

- **Amazingly Powerful**!
- After results are tokenized, change them!
  - remove/add information
  - convert to other types
  - use as object initializers
  - if you can think of it, you can probably do it
- 3 Flavours (mmm…) of parse action function:
  - f(t); t ⇨ list of parsed tokens
  - f(l, t); l ⇨ location in parsed string
  - f(s, l, t); s ⇨ the parsed string

Parse Actions continued

- To use:
  - define function for parse action
  - add to parse expression:
    - `addParseAction()`
      - allows sequential, multiple actions
    - `setParseAction()`
      - defines a single action (can add more)

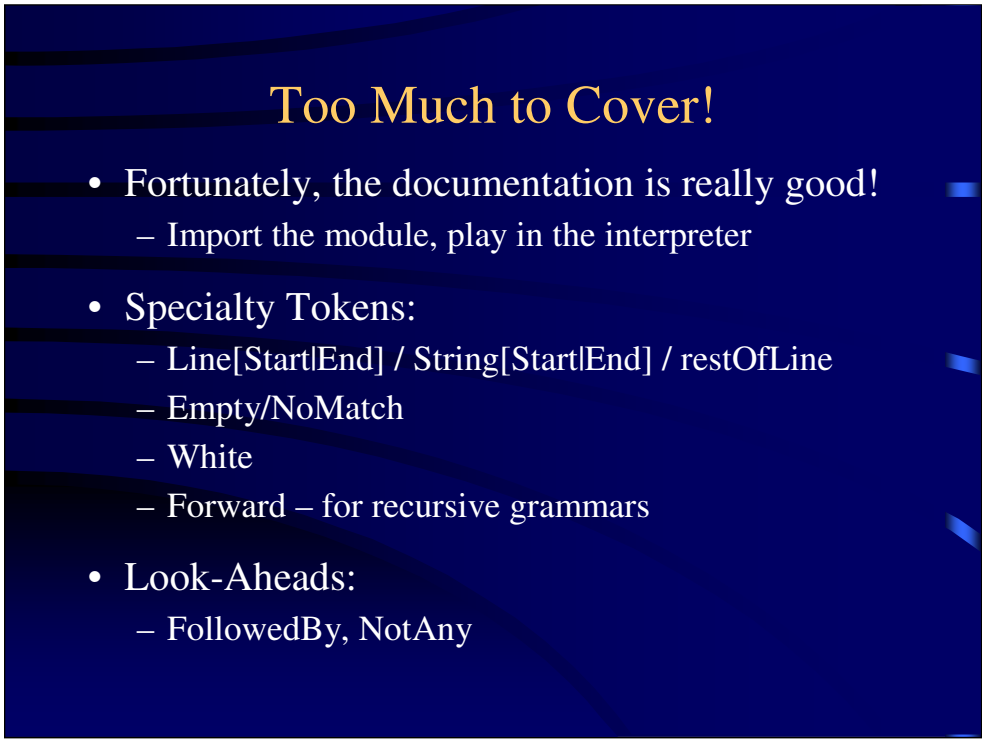
```
IPnum = Word("012346789", max=3)
def checkIPNum(toks):
  i = int(t[0])
  if i > 255: raise IPError
  return [i]
IPNum.setParseAction(checkIPNum)
```

Parse Actions continued, again

- Several built-in helper actions:
  - `removeQuotes()`
  - `replaceWith(replStr)`
  - … to name 2

Results-Driven

- parse*() functions return ParseResult objects
- Allow access to parse results (duh):
  - as a list (len, indexing, etc.)
  - as a dict
    - (only for results named with `setResultsName()`)
  - as attributes (for named results)
- For some applications, a ParseResults object may not be quite right, so…
  - `asList()`, `asDict()` methods return those types
- `asXML()` – results as an XML string



Too much to cover in 45 minutes!

Text/Example for Forward:

use to create recursive grammars

declare blank 'Forward' object

insert 'real' token with << operator

```
listDef = Forward()
listElem = <other things> | listDef
listDef << Group(Suppress("[") + ZeroOrMore(listElem) +
Suppress("]"))
```

WAY Too Much to Cover!

- Even more methods!
  - `leaveWhitespace()`
  - `parseWithTabs()`
  - `setFailAction()`
  - `setDebug() / setDebugActions()`
- Pre-fab tokens/expressions!
  - dblQuotedString
  - [language]StyleComment
- Pre-fab strings!
  - alphas, alphanums, alphas8bit, nums, hexnums…

leaveWhitespace(self)
Disables the skipping of whitespace before matching the characters in the ParserElement's defined pattern.

parseWithTabs(self)
Overrides default behavior to expand <TAB>s to spaces before parsing the input string.

setFailAction(self, fn)
Define action to perform if parsing fails at this expression.

setDebug(self, flag)
Enable display of debugging messages while doing pattern matching.

setDebugActions(self, startAction, successAction, exceptionAction)
Enable display of debugging messages while doing pattern matching.

pyparsing

A User's Perspective

Wilson Fowlie, Delta Controls, Inc.
wfowlie@deltacontrols.com