Building OT Grammars in PyPhon
A Tutorial for the Stanford P-Interest Workshop

Sam Bowman

March 16, 2012
What PyPhon is

- PyPhon is a Python Toolkit for constraint based grammars (OT and HG) with a rudimentary command-line interface to build grammars and typologies.
- Primarily the work of Jason Riggle and Max Bane at the [University of] Chicago Language Modeling Lab. (I’ve recently been somewhat involved.)
PyPhon is a Python Toolkit for constraint based grammars (OT and HG) with a rudimentary command-line interface to build grammars and typologies.

Primarily the work of Jason Riggle and Max Bane at the [University of] Chicago Language Modeling Lab. (I’ve recently been somewhat involved.)

**The Grand Vision:** To enable phonologists to quickly develop large-scale constraint-based grammars in a way that is **guaranteed** to be:

- Accurate (no erroneous tableaux)
- Precise (all constraints and feature values clearly described)
- Monster-free (the typological predictions of a model—including quantitative and implicational generalizations—are immediately visible)
What PyPhon Can Do

To quote the documentation, PyPhon currently offers the following functionality:

- State your model as a set of hand-crafted tableaux, or simply specify your constraints as regular expressions (finite-state acceptors), and let full tableaux be computed automatically for any input forms.
What PyPhon Can Do

To quote the documentation, PyPhon currently offers the following functionality:

- State your model as a set of hand-crafted tableaux, or simply specify your constraints as regular expressions (finite-state acceptors), and let full tableaux be computed automatically for any input forms.
- Tableaux candidates can be generated for either OT or HG.
What PyPhon Can Do

To quote the documentation, PyPhon currently offers the following functionality:

- State your model as a set of hand-crafted tableaux, or simply specify your constraints as regular expressions (finite-state acceptors), and let full tableaux be computed automatically for any input forms.
- Tableaux candidates can be generated for either OT or HG.
- Produce finite state diagrams of your regular constraints.
What PyPhon Can Do

To quote the documentation, PyPhon currently offers the following functionality:

- State your model as a set of hand-crafted tableaux, or simply specify your constraints as regular expressions (finite-state acceptors), and let full tableaux be computed automatically for any input forms.
- Tableaux candidates can be generated for either OT or HG.
- Produce finite state diagrams of your regular constraints.
- Generate optima and/or typologies in both OT and HG, optionally subject to grammatical restrictions that can be specified in a variety of ways:
  - As stratified hierarchies (OT)
  - As partial orders (OT)
  - As sets of elementary ranking conditions (OT)
  - As sets of weight assignments and/or inequalities (HG)

...
What PyPhon Can Do

...  

- Count the number of rankings (OT), or volume of weightings (HG), that generate each output form.
What PyPhon Can Do

... 

- Count the number of rankings (OT), or volume of weightings (HG), that generate each output form.
- Compute typological implications (T-Orders; after Anttila and Andrus 2006) and render pretty graphs.
What PyPhon Can Do

...  

- Count the number of rankings (OT), or volume of weightings (HG), that generate each output form.
- Compute typological implications (T-Orders; after Anttila and Andrus 2006) and render pretty graphs.
- Create and search typologies in cyclic OT and two-tiered LPM-OT (after Kiparsky, 2000).
All three can calculate typologies for both ranked or weighted constraints.
The Alternatives (corrections welcome)

- All three can calculate typologies for both ranked or weighted constraints.
- OT-Help 2 (UMass team, including Chris Potts):
  - Well documented; great for finding ranking conditions.
  - Does Harmonic Serialism as well as OT/HG.
  - Automatically finds contenders only for Harmonic Serialism, not OT/HG; requires the user to specify the operations in GEN.
The Alternatives (corrections welcome)

- All three can calculate typologies for both ranked or weighted constraints.

- **OT-Help 2 (UMass team, including Chris Potts):**
  - Well documented; great for finding ranking conditions.
  - Does Harmonic Serialism as well as OT/HG.
  - Automatically finds contenders only for Harmonic Serialism, not OT/HG; requires the user to specify the operations in Gen.

- **OTSoft 2 (Hayes; UCLA):**
  - More explicitly designed to facilitate grammar development: Clearly notes redundant constraints and harmonically bounded (impossible) candidates.
  - Generates T-orders and Hasse diagrams.
  - Written in a dead, Windows-only, software framework (VB6). Likely moribund.
A constraint is represented as a set of regular expressions which accept (match) any local input–output mappings that violate it.

PyPhon has its own regular expression syntax, which includes special symbols for context, for insertion/deletion and for correspondence with input forms.

No comprehensive documentation: Look to example files. (Or ask me!)
MaxV:
_[-cons]:-

IO-IDENT[high]:
_ [+high]: [-high]_
_ [-high]: [+high]_

*NV_{oral}:
[+nasal]_[-cons, -nasal]_

*P-SHAPED:
_(b|p|d|q)_

*CC&DEPC:
C_:-:C_
_:-:C_C
Local markedness and faithfulness constraints of all sorts are easy to write and fast to run.
Local markedness and faithfulness constraints of all sorts are easy to write and fast to run.

Complex long-distance markedness constraints and non-local constraint conjunction are possible, but tend to be somewhat unwieldy, and can cause explosive growth in run time.
Local markedness and faithfulness constraints of all sorts are easy to write and fast to run.

Complex long-distance markedness constraints and non-local constraint conjunction are possible, but tend to be somewhat unwieldy, and can cause explosive growth in run time.

Constraints which can accrue violations faster than linearly in the length of the string cannot be represented. This includes the already theoretically tenuous ALIGN(FOOT, WD).
In Practice: PyPhon File Types

- **Inputs file (.csv):** A list of input forms, in your representation of choice, one per line, with comments marked with #
In Practice: PyPhon File Types

- **Inputs file (.csv):** A list of input forms, in your representation of choice, one per line, with comments marked with #

- **Model description file (.csv):** A file specifying (1) the feature bundles associated with which segments in your representation and (2) the regular expressions that define your constraints.
In Practice: PyPhon File Types

- **Inputs file** (.csv): A list of input forms, in your representation of choice, one per line, with comments marked with #

- **Model description file** (.csv): A file specifying (1) the feature bundles associated with which segments in your representation and (2) the regular expressions that define your constraints.

- **Model file** (.mod): A non-human-readable representation of Gen and Eval, generated from your model description file by makemodel.
In Practice: PyPhon File Types

- **Inputs file (\.csv)**: A list of input forms, in your representation of choice, one per line, with comments marked with #

- **Model description file (\.csv)**: A file specifying (1) the feature bundles associated with which segments in your representation and (2) the regular expressions that define your constraints.

- Model file (\.mod): A non-human-readable representation of Gen and Eval, generated from your model description file by makemodel.

- Grammar file (\.csv): A set of stipulations about the rankings or weightings of your model’s constraints.
In Practice: PyPhon File Types

- **Inputs file** (.csv): A list of input forms, in your representation of choice, one per line, with comments marked with #

- **Model description file** (.csv): A file specifying (1) the feature bundles associated with which segments in your representation and (2) the regular expressions that define your constraints.

- **Model file** (.mod): A non-human-readable representation of \texttt{GEN} and \texttt{EVAL}, generated from your model description file by \texttt{makemodel}.

- **Grammar file** (.csv): A set of stipulations about the rankings or weightings of your model’s constraints.

- **Tableaux file** (.csv): A set of unranked tableaux specifying all of the contenders for each input, and all of their constraint violation profiles.
Acceptor graph file (.dot): A graph file containing a finite-state acceptor representation of a single constraint or of all of EVAL. Used for both visualization and caching. Editable in a pinch.
In Practice: PyPhon File Types

- Acceptor graph file (.dot): A graph file containing a finite-state acceptor representation of a single constraint or of all of EVAL. Used for both visualization and caching. Editable in a pinch.

- T-order file (.dot): A graph file representing implicational relationships between input-output mappings.
In Practice: PyPhon File Types

- Acceptor graph file (.dot): A graph file containing a finite-state acceptor representation of a single constraint or of all of EVAL. Used for both visualization and caching. Editable in a pinch.

- T-order file (.dot): A graph file representing implicational relationships between input-output mappings.

- Typology file (.csv): Shows the possible languages corresponding to a set of tableaux and their corresponding ERCs and R-volumes.
In Practice: PyPhon File Types

- Acceptor graph file (.dot): A graph file containing a finite-state acceptor representation of a single constraint or of all of EVAL. Used for both visualization and caching. Editable in a pinch.

- T-order file (.dot): A graph file representing implicational relationships between input-output mappings.

- Typology file (.csv): Shows the possible languages corresponding to a set of tableaux and their corresponding ERCs and R-volumes.

- (!) Input-output file (.csv): Used to test for the existence of a language, as specified by a set of input-output pairs.

- (!) Two tier languages file (.csv): Similar to a typology file, only with less information, and corresponding to a stratal grammar.

Sam Bowman  Building OT Grammars in PyPhon
In Practice: The Scripts

- `pyphon_makemodel.py`: Compiles a model description file into a model file. Optionally also generates acceptor graphs.
In Practice: The Scripts

- `pyphon_makemodel.py`: Compiles a model description file into a model file. Optionally also generates acceptor graphs.

- `pyphon_maketableaux.py`: Takes a model file, an inputs file, and a grammar option (OT or HG), and generates full tableaux for each input.
In Practice: The Scripts

- **pyphon_makemodel.py**: Compiles a model description file into a model file. Optionally also generates acceptor graphs.
- **pyphon_maketableaux.py**: Takes a model file, an inputs file, and a grammar option (OT or HG), and generates full tableaux for each input.
- **pyphon_generate.py**: Takes a tableaux file, a grammar option (OT or HG) and optionally a grammar file, and generates typologies and T-orders.
In Practice: The Scripts

- **pyphon_makemodel.py**: Compiles a model description file into a model file. Optionally also generates acceptor graphs.

- **pyphon_maketableaux.py**: Takes a model file, an inputs file, and a grammar option (OT or HG), and generates full tableaux for each input.

- **pyphon_generate.py**: Takes a tableaux file, a grammar option (OT or HG) and optionally a grammar file, and generates typologies and T-orders.

- **pyphon_twotier.py**: Takes a model file and an inputs file, and generates tableaux and typologies for two-tiered (word and postlexical) LPM-OT.
Get a copy of the source package (as a ZIP or over Subversion) from: http://code.google.com/p/clml/

- cd into the pyphon directory and run: python setup.py install
- Optional: For HG typologies, get GLPK (link from the CLML site).
Writing Inputs and Constraints

(switch to the software itself)
Tips, in lieu of complete documentation

- If you use any filters on your outputs, make sure that they are not too strict to rule out any potentially interesting candidates.
Tips, in lieu of complete documentation

- If you use any filters on your outputs, make sure that they are not too strict to rule out any potentially interesting candidates.

- If you are having trouble getting a grammar to load, try adding a blank grammar to the end. The last grammar in a file may behave oddly.
Tips, in lieu of complete documentation

- If you use any filters on your outputs, make sure that they are not too strict to rule out any potentially interesting candidates.

- If you are having trouble getting a grammar to load, try adding a blank grammar to the end. The last grammar in a file may behave oddly.

- Delete your model file and re-build it if something is refusing to change. Also, make sure that everything made it into the inventory list and \texttt{CON} that you want to have there.
Tips, in lieu of complete documentation

- If you use any filters on your outputs, make sure that they are not too strict to rule out any potentially interesting candidates.
- If you are having trouble getting a grammar to load, try adding a blank grammar to the end. The last grammar in a file may behave oddly.
- Delete your model file and re-build it if something is refusing to change. Also, make sure that everything made it into the inventory list and CON that you want to have there.
- Don’t use file extensions on the command line. PyPhon will fill them in as needed.
Tips, in lieu of complete documentation

- If you use any filters on your outputs, make sure that they are not too strict to rule out any potentially interesting candidates.

- If you are having trouble getting a grammar to load, try adding a blank grammar to the end. The last grammar in a file may behave oddly.

- Delete your model file and re-build it if something is refusing to change. Also, make sure that everything made it into the inventory list and `CON` that you want to have there.

- Don’t use file extensions on the command line. PyPhon will fill them in as needed.

- If you forget the arguments for one of the basic PyPhon scripts, just invoke it with no arguments. It’ll tell you.
Tips, in lieu of complete documentation

➤ If you use any filters on your outputs, make sure that they are not too strict to rule out any potentially interesting candidates.

➤ If you are having trouble getting a grammar to load, try adding a blank grammar to the end. The last grammar in a file may behave oddly.

➤ Delete your model file and re-build it if something is refusing to change. Also, make sure that everything made it into the inventory list and CON that you want to have there.

➤ Don’t use file extensions on the command line. PyPhon will fill them in as needed.

➤ If you forget the arguments for one of the basic PyPhon scripts, just invoke it with no arguments. It’ll tell you.

➤ Let me know if something doesn’t work. It might be a bug.
What is still in the works?

- A manual
- Faster operation
- Full support for Unicode characters
- Direct output to \LaTeX
- A chocolate and wine icebox cake! (Made while compiling this presentation. May not be uploaded to the lab site. May be available in the kitchen at five o’clock.)
- A cleaner, more consistent interface (maybe even a GUI)
- A fully functioning, three-tiered model of LPM-OT
- Whatever you want...
Questions and comments:
  ▶ sbowman@stanford.edu
  ▶ http://clml.uchicago.edu
Pretty Graphs!

/Co#Co/ \rightarrow [Co#Co] (0,0,0,0)  
/CoCu/ \rightarrow [CoCu] (0,0,0,0)  
4 languages.

/I.a#CoCuo#/ \rightarrow [u.u#CoCuo#] (0,1,2,0)  
/a#CoCuo#/ \rightarrow [u#CoCuo#] (0,1,1,0)  
1 language.

/I.a#CoCuo#/ \rightarrow [I.a#CoCuo#] (1,0,0,0)  
/a#CoCuo#/ \rightarrow [a#CoCuo#] (1,0,0,0)  
1 language.

/I.a#CoCuo#/ \rightarrow [I.a#CaCIa#] (0,0,3,0)  
/a#CoCuo#/ \rightarrow [a#CaCIa#] (0,0,3,0)  
1 language.

/I.a#CoCuo#/ \rightarrow [u.o#CoCuo#] (0,0,2,1)  
/a#CoCuo#/ \rightarrow [o#CoCuo#] (0,0,1,1)  
1 language.