Factotum

Andrew Shin

2012

Introduction

Factotum is an open source project originally conceived by Robert Uzgalis at UCLA. Its primary goal is to aid the study of humanity, religion, linguistics and so forth, which deal with a large amount of facts that need to be organized, so that the user can easily validate a topic of questions based on the known facts. Claire Abu-Hakima has written the code that corresponds to parsing and lexing aspects of Factotum under supervision of Paul Eggert, and I would like to extend it, based on her work, with emphasis on interface aspects. While Abu-Hakima's work used linguistic data collected from Wikipedia to test it out, I did not use her data due to different marker usage and several other issues. Please refer to the original Factotum repository for details on parsing, lexing, and how she made use of Wikipedia language data (www.github.com/claireabu/Factotum). New codes were written in Python 3 and the original codes by Claire have also been ported into Python 3 and behave the same as before for the most parts. All codes are available on www.github.com/shinysup/Factotum.

Python 3

In order to be able to work with any language, the original codes written in Python 2 have been ported to Python 3 so that it can work In UTF-8 instead of ASCII. Note that you still need to specify the encoding scheme when opening file, as follows:

```
myfile = open(filename, 'r', encoding='utf-8')
```

Then, one will be able to run the codes with non-English characters (of course, as long as it is in Unicode).

Porting process was done by both automation and manual modification. 2to3, which is a built-in conversion tool included in Python 2.6, was used to automate the conversion to Python 3. Note that 2to3 is only available on Python 2. Only 3to2, which does the opposite job, is offered in Python 3. You can run 2to3 on Factotum folder with the following command:

```
2to3 –w –f all –f idioms Factotum
```

2to3 can be applied to both individual files or a folder. *–w* flag writes the changes to the file, *-f* flag runs the predefined set of fixers. *all* enables all default fixers, and *idioms* is a fixer that is not run by default.

It is important to note that 2to3 is far from solving every problem in the conversion process. 2to3 is highly limited to changing the old syntax to the new ones. For example, it changes *print 'some string'* to *print('some string'),* which is a function in Python 3. 2to3 does not fix many other issues such as indentation, module behaviors, etc.

In Python 2, ambiguous indentation style was allowed. In Python 3, a strict indentation with tab or a fixed number of spaces is enforced, and 2to3 does not fix this, which leads to errors. One can run a program to replace blocks of spaces with tabs, but a manual observation is still almost essential.

Also, some of the constants are different. For example, in Python 2, from the following command,

>>> from string import *

>>> letters

we get the set of characters defined by regular excpression,

'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRXTUVWXYZ'

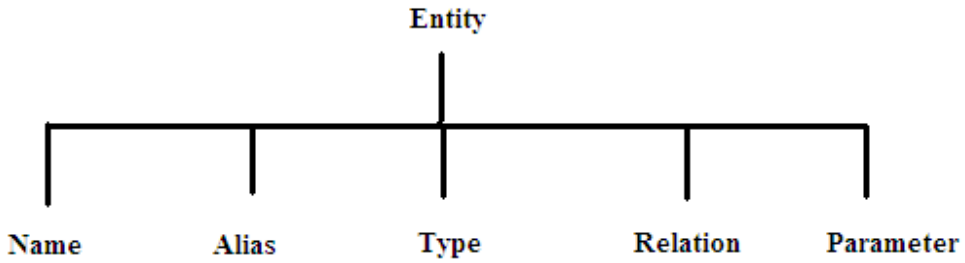
But in Python 3, with the same command, we get

NameError: name 'letters' is not defined

This happens because the constant *letters* from *string* module is now called *ascii_letters* in the same module.

Fact

The data structure for the fact part is constructed in the following way. Each entity consists of five parameters; name, aliases, types, relations, and parameters. Name must be unique to entity, so it is a string. Types and aliases can have multiple values, so they are lists of strings. Relations and parameters also have multiple values, but they are treated differently because they consist of sub-parameters within themselves. Relation consists of its unique relation name, its print format, and its object if any. If a relation does not have an object, we know it is an attribute. Parameters are those that contain numeric values. It thus consists of its unique name, its unique numeric value, and units. Because they have parameters themselves, relations and parameters are constructed as dictionaries, with each parameter being their keys. Note that type, relation, and parameter, despite being parts of fact, are constructed from imported model classes. Finally, the entities dictionary consists of each entity as its key with its parameters as values. Following functions are designed to help the user easily manipulate the data structure in the interface.



init_entity(ename):

This initializes the entity dictionary (note that it is a key in *entities* dictionary), sets its name as *ename*, and initializes other parameters with empty lists (or empty dictionaries).

add_param(ename,pname,val):

For a parameter *pname* of entity *ename*, the new value *val* is added. If the parameter *pname* is alias, we can just append the new value since they are just list of strings. If the parameter *pname* is type, we check whether the type *val* exists in types dictionary of the model. If it does, we can append it to our type list. Otherwise, we print error message. If the parameter *pname* is a relation or a parameter, we call *init_relation* or *init_parameter* which does the checking from the model and key assigning. Adding name can only be done when the name value is an empty string. This rarely happens because name gets set up when the entity gets initialized, but user may use *del_param* to delete the name value, which is the only possible case for having an empty string for name. It prints out an error message if entity name or parameter name is not defined.

modify_param(ename,pname,oval,nval):

This replaces the pre-stored value *oval* with a new value *nval* for a parameter *pname* of entity *ename*. Since aliases and types are lists of strings, we simply remove the *oval* and append the *nval*.

For relations and parameters, we call model API and run their init functions. This is because we can only modify the relation name (or parameter name) from fact. If we were to modify sub-parameters of relation, we must do so from the model.

For name, we simply assign a new string. If entity name or parameter name is not defined, or if the old value is not present, it prints out the error message.

del_param(ename,pname,val):

This deletes the value *val* from parameter *pname* of entity *ename*. For aliases and types, it removes the value from the string list. For relations and parameters, it deletes the entire key from the dictionary that corresponds to the value. For name, it assigns an empty string. It prints out an error message if entity name or parameter name is not defined.

get_value(ename,pname):

This returns the string or a list or a dictionary corresponding to the parameter *pname* of entity *ename*.

show_all():

This simply prints the entire *entities* dictionary for reading purpose.

init_relations(ename,rel):

init_parameters(ename,param):

These two functions are not directly used by the user, but are called from other functions, such as *add_param*, to initialize the relation a parameter dictionaries. We check whether *rel* or *param* exists in relations or parameters dictionary in the model. If yes, we initialize a new sub-dictionary as a key to relation or parameter dictionary. If not, print error message. Note that each key of relation or parameter is copied from the model.

A simple example run:

```
e = 'testEnt'
init_entity(e)
f = 'testTyp'
add_param(e,'Types',f)
g = get_value(e,'Alias')
print(g)
h =  'newTyp'
modify_param(e,'Types',f,h)
show_all()
```

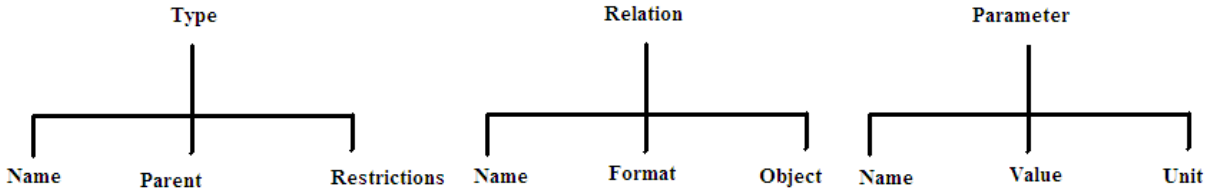
should print

```
type not defined
value testTyp not in type dictionary
{'testEnt': {Alias': [], 'Relations': [], 'Parameters': [], 'Name': 'testEnt', 'Types': []}}
```

(assuming that *testTyp* and *newTyp* are not yet in types dictionary, which must be added using model API).

Model

The API for the model part is similar to the fact part, except that it deals with types, relations, and parameters, instead of entities. Each type consists of name, a set of restrictions applied to it, and its parent type if any. Type name and parent type are unique, and are strings. Restrictions are a list of strings, but we need a way for Python to interpret the restriction so it can test/apply the type restrictions on the entity. One possibility is making it a boolean expression that Python can understand and apply. Relations are any attributes that describe something about an entity. It has a unique name, and a print format for output, and may or may not have an object. Parameters are nearly the same as relations except they contain numeric values. It has its unique name, its unique numeric value, and unique unit.



1) class *typesClass*:

init_type(tname):

This initializes the type dictionary (as a key in *types* dictionary), sets its name as *tname*, its parent as an empty string, and initializes restrictions as a list of string.

add_restriction(type,res):

This appends the new restriction *res* to the restriction list.

set_parent(type,ptype):

This sets the parent as *ptype*. It does not append to the original value because parent type is unique. Note that we can use this function to modify parent, since modification of parent essentially means re-setting the parent and assigning it to a string. For that reason, *modify_parent* function is not included in the API whereas *modify_restriction* is (because it deals with a list of strings and needs to remove and append).

del_restriction(type,res):

This removes the restriction *res* from the restriction list.

del_parent(type):

This sets the parent type as an empty string.

modify_restriction(type,ores,nres):

This function works by calling two of the functions above; It first calls *del_restriction* to remove the original restriction *ores*, and then calls *add_restriction* to append new restriction *nres*.

return_keys():

returns the list of keys of the global dictionary *types*

get_value(type,param):

This returns the string or a list corresponding to the parameter *param* of type *type*.

show_all():

This prints the entire *types* dictionary for reading purpose.

2) class *relationsClass*:

init_relation(rname):

initializes the relation dictionary (as a key in *relations* dictionary), sets its name as *rname*, its format as an empty string, and objects as an empty list of string.

set_format(rname,format):

If *rname* exists in *relations* dictionary, it sets *format* as its format as a string. Otherwise, print an error message.

add_object(rname,obj):

If *rname* exists in *relations* dictionary, it appends *obj* to the object list. Otherwise, print an error message.

del_format(rname)

If *rname* exists in relations dictionary, this sets format as an empty string.

del_object(rname,obj):

If *rname* exists in *relations* dictionary, it removes *obj* from the object list.

modify_object(rname,oobj,nobj):

It calls *del_object* and *add_object* in series to remove *oobj* from the object list and append *nobj* to the object list.

get_value(rname,param):

This returns the string or a list corresponding to the parameter *param* of relation *rname*.

show_all():

This prints the entire *relations* dictionary.

return_keys():

returns the list of keys of the global *relations* dictionary.

3) class *parametersClass*:

init_parameter(pname):

initializes the parameter dictionary (as a key in *parameters* dictionary), sets its name as *pname*, its value and uint as empty string.

set_value(pname,val):

If *pname* exists in *parameters* dictionary, this sets the value string as *val*. Otherwise, print an error message. This also performs the function of modifying the value, since value is just a string that we can repeatedly set.

set_unit(pname,unit):

If *pname* exists in *parameters* dictionary, this sets unit string as *unit*. Otherwise, print an error message. This also performs the function of modifying the unit, since unit is a string that we can repeatedly set.

del_value(pname):

If *pname* exists in *parameters* dictionary, this sets value string as an empty string.

del_unit(pname):

If *pname* exists in parameters dictionary, this sets unit string as an emptry string. This returns the string corresponding to the parameter *param* of parameter *pname*.

show_all():

This prints the entire *parameters* dictionary.

return_keys():

returns the list of keys of the global *parameters* dictionary.

Another thing to note is that there is a *patterns* dictionary, which is shared by *relations* and *parameters*. Each key of patterns dictionary has either relation or parameter as its value, indicating which one the pattern belongs to.

Note that Claire's codes have been modified so that they work in accordance with fact.py and model.py. For example, they import fact and model and instead of just having *types = {}*, they now have *types = model.typesClass()*. Also, for example, instead of setting the parameters for *self.types* by doing *self.types[t][param]={}*, we now just call the function *init_type* from model.py, as in *self.types.init_type(t)*. There are also minor differences between Claire's original codes and the modified ones, such as different ways to print output.

Validation

validate.py was originally written by me as an interface for checking and validating of newly entered facts based on pre-defined facts. It originally had a different data structure from fact and model described above, so I adjusted so that it becomes compatible with current fact and model data structure. As a result, many parts of the code are now commented out and some parts have become meaningless but I have left it in repository in the hope that it might be useful in future development of checking interface. Originally, it collected data from the text file generated by running mkvocab.py. Now it imports fact and model. Functions such as *readVocab(), collectRelations(), updateDict()* were used for the purpose of collecting data from text file, but they are now commented out and can be ignored. Still, there are functions that have potential to be useful.

getTypeof() simply returns the list of possible types of an argument entity.

hasNumeric() and *returnNumeric()* are made to differentiate parameters from relations. *hasNumeric()* is a boolean function that returns true if an argument contains a numeric in it. *returnNumeric()* retuns the numeric values within the argument as a list. It returns a list instead of a numeric value because there might be more than one numeric value in the argument. The numeric values in the list are later replaced by () in the parameter so that we can store parameter and corresponding value separately. Since we now have separate relations and parameters dictionaries in the model, these functions have become outdated, but I have left it.

validateNewFact() is where the user enters new facts and gets a check whether it is a valid fact. First, the user enters a new fact according to the enforced grammar rule. In current grammar rule, the first token of the input can start with either a name of subject or a type, indicated by []. So we first check whether the first token is a subject or a type by checking the presence of []. If it is a subject, then we check whether the rest of the input has to do with type information, such as type, restriction, or parent. All of them are distinguished by distinct markers, so we check for the presence of each marker. If none of the markers is present, it means the input is a relation (or parameter). We use the hasNumeric() function from above to check whether it is relation or parameter. If parameter, we replace the numeric values with () and puts the values separately for corresponding parameter key.

If we do find a marker, then we know by looking at the marker what element it is, so we get to checking part. If the subject is entered with a type and the subject is not in our fact dictionary, then we ask if the subject satisfies the type restriction. If it passes all restrictions, then it is validated and added to the dictionary. Otherwise, it is not validated. Checking the parent and restrictions for a type is straightforward, done just by looking at the dictionary's corresponding key.

Grammar

A user can enter a new fact, entity, or parameter information using fact and model API, but can also create and modify a text file. If using a text file, the following grammar expressions are enforced. The same rule applies when the user needs to modify the __.f file.

subject predicate (where subject is a key of entities dictionary)
entity [type]
[type] >>parent type
[type] #restriction

The original codes by Claire have also been modified to meet these rules. I have not set up the marker and expression for aliases yet.

Suppose we have the following predefined facts:

Andrew [Korean]
Ichiro [Japanese]
[Korean] >>Asian
[Japanese] >>Asian
[Korean] #isBornInKorea
[Japanese] #isBornInJapan
Andrew is male
Ichiro plays baseball
Ichiro hit 15 home runs in 2009

If any of the predefined facts is entered as a new fact, then it gets validated as it should. If, in a new fact, a new entity is said to be of one of the predefined types, the program asks whether the new entity qualifies for restrictions of that type. If the answer is yes, then the new entity is added to the type's entity list. If not, then it is not validated. However, if a new fact involves a new type that is not in the predefined facts, then we cannot check whether the statement is valid. It currently just gives dictionary key error, and I will fix it to give a proper warning message. If a new type were to be added to our facts, then the user should modify the ____.f file and make new vocabularies out of it.

Andrew [Korean] → validated
David [Korean] → Is David born in Korea? (y/n) → validated → new entity list
Andrew [Chinese] → keyerror

Likewise, pre-defined parent relations between types are validated, but if child type is not defined, it gives keyerror. If parent type is not defined previously but the child type is, then we know the statement is not right. Relations work in the same manner.

[Korean] >>Asian → validated
[Japanese] >>Japanese → not validated
[Korean]>>European → not validated
[Chinese] >> Asian → keyerror
Ichiro plays baseball → validated
Ichiro hit 6 home runs in 2009 → not validated

Further Work

Focus of my work so far has been getting it working in Python 3, settling the data structure, and setting the basis for its API. One difficulty that arises when working on Factotum is the highly abstract nature of the project and the ambiguity of data structures. There are many possible ways we can handle data structure, parameters, APIs, which require much study and deliberation over appropriate ways to implement them. I have built the structure described above differently from Claire's original (have modified her codes accordingly), based on professor Uzgalis' suggestions. I would suggest that one sticks to the data structure above, so we can have the issue settled rather than having more variants, but it is complete up to the programmer.

As mentioned above, restrictions for types are currently a list of strings, and we need a way to represent them so that Python can interpret it and apply it to entities of each type. One possibility is to use Boolean expression, and another option is to write a programming language that Python can interpret.

Also, patterns dictionary for relation and parameter must be handled with APIs. Remember that relations and parameters share the same *patterns* dictionary, and each pattern (which is a key in *patterns* dictionary) is distinguished by its value as either a relation or a parameter. Handling multiple numeric values for parameter may also be desired for optimization.

Reference

Beazley, David M. Python: Essential Reference 4th Edition. Addison Wesley, 2008

Uzgalis, Robert C. "Factotum III: Reference Manual." Unpublished manuscript, Los Angeles, California, 2011

http://docs.python.org/release/3.0.1/tutorial/

http://docs.python.org/library/2to3.html

https://github.com/claireabu/Factotum