pymorphy2 Documentation

Выпуск v0.1

Mikhail Korobov

19 October 2013

Оглавление

pymorphy2 написан на языке Python (работает под 2.x и 3.x). Он умеет:

1. приводить слово к нормальной форме (например, "люди -> человек", или "гулял -> гулять").
2. ставить слово в нужную форму. Например, ставить слово во множественное число, менять падеж слова и т.д.
3. возвращать грамматическую информацию о слове (число, род, падеж, часть речи и т.д.)

При работе используется словарь OpenCorpora; для незнакомых слов строятся гипотезы. Библиотека достаточно быстрая: в настоящий момент скорость работы - от нескольких тыс слов/сек до > 100тыс слов/сек (в зависимости от выполняемой операции, интерпретатора и установленных пакетов); потребление памяти - 10...20Мб; полностью поддерживается буква ё.

Лицензия - MIT.

Глава 1

Содержание

1.1 Документация

Важно: в примерах используется синтаксис Python 3.

Важно: в примерах используется синтаксис Python 3.

1.1.1 Руководство пользователя

Установка

Для установки воспользуйтесь pip:

```
pip install pymorphy2
pip install pymorphy2-dicts
pip install 'DAWG-Python >= 0.7'
```

pymorphy2-dicts - это пакет со словарями OpenCorpora, скомпилированными в формат pymorphy2.

Если вы используете CPython (не PyPy), и в системе есть компилятор и т.д., то вместо DAWG-Python можно установить библиотеку DAWG, которая позволит pymorphy2 работать быстрее:

```
pip install 'DAWG >= 0.7'
```

Морфологический анализ

Морфологический анализ - это определение характеристик слова на основе того, как это слово пишется. При морфологическом анализе не используется информация о соседних словах.

В pymorphy2 для морфологического анализа слов (русских) есть класс MorphAnalyzer.

```
>>> import pymorphy2
>>> morph = pymorphy2.MorphAnalyzer()
```

Экземпляры класса MorphAnalyzer обычно занимают порядка 10-15Мб оперативной памяти (т.к. загружают в память словари, данные для предсказателя и т.д.); старайтесь организовать свой код так, чтобы создавать экземпляр MorphAnalyzer заранее и работать с этим единственным экземпляром в дальнейшем.

С помощью метода MorphAnalyzer.parse() можно разобрать слово:

```
>>> morph.parse('стали')
[Parse(word='стали', tag=OpencorporaTag('VERB,perf,intr plur,past,indc'), normal_form='стать', score=0.983766, methods_stack=
 Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,gent'), normal_form='сталь', score=0.003246, methods_stack=((
 Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,datv'), normal_form='сталь', score=0.003246, methods_stack=((
 Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn sing,loct'), normal_form='сталь', score=0.003246, methods_stack=((
 Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn plur,nomn'), normal_form='сталь', score=0.003246, methods_stack=
 Parse(word='стали', tag=OpencorporaTag('NOUN,inan,femn plur,accs'), normal_form='сталь', score=0.003246, methods_stack=((
```

Примечание: если используете Python 2.x, то будьте внимательны - юникодные строки пишутся как u'стали'.

В этом примере слово "стали" может быть разобрано и как глагол ("они стали лучше справляться"), и как существительное ("кислородно-конверторный способ получения стали"). На основе одной лишь информации о том, как слово пишется, понять, какой разбор правильный, нельзя, поэтому анализатор может возвращать несколько вариантов разбора.

У каждого разбора есть нормальная форма, которую можно получить, обратившись к атрибутам normal_form или normalized:

```
>>> p = morph.parse('стали')[0]
>>> p.normal_form
'стать'
>>> p.normalized
Parse(word='стать', tag=OpencorporaTag('INFN,perf,intr'), normal_form='стать', score=1.0, methods_stack=((<DictionaryAnaly
```

Примечание: См. также: Постановка слов в начальную форму.

Кроме того, у каждого разбора есть тег:

```
>>> p.tag
OpencorporaTag('VERB,perf,intr plur,past,indc')
```

Тег - это набор граммем, характеризующих данное слово. Например, тег 'VERB,perf,intr plur,past,indc' означает, что слово - глагол (VERB) совершенного вида (perf), непереходный (intr), множественного числа (plur), прошедшего времени (past), изъявительного наклонения (indc).

См. также: Обозначения для граммем.

pymorphy2 умеет разбирать не только словарные слова; для несловарных слов автоматически задействуется предсказатель. Например, попробуем разобрать слово "бутявковедами" - pymorphy2 поймет, что это форма творительного падежа множественного числа существительного "бутявковед", и что "бутявковед" - одушевленный и мужского рода:

```
>>> morph.parse('бутявковедами')
[Parse(word='бутявковедами', tag=OpencorporaTag('NOUN,anim,masc plur,ablt'), normal_form='бутявковед', score=1.0, methods
```

Работа с тегами

Для того, чтоб проверить, есть ли в данном теге отдельная граммема (или все граммемы из указанного множества), используйте оператор in:

```
>>> 'NOUN' in p.tag  # то же самое, что и {'NOUN'} in p.tag
False
>>> 'VERB' in p.tag
True
>>> {'VERB'} in p.tag
True
>>> {'plur', 'past'} in p.tag
True
>>> {'NOUN', 'plur'} in p.tag
False
```

Примечание: В Python 2.6 не поддерживается {'NOUN', 'plur'} синтаксис для задания множеств. Если у вас Python 2.6, то тут и дальше в примерах используйте форму записи set(['NOUN', 'plur']).

Кроме того, у каждого тега есть атрибуты, через которые можно получить часть речи, число и другие характеристики:

```
>>> p.tag.POS           # Part of Speech, часть речи
'VERB'
>>> p.tag.animacy       # одушевленность
None
>>> p.tag.aspect        # вид: совершенный или несовершенный
'perf'
>>> p.tag.case          # падеж
None
>>> p.tag.gender        # род (мужской, женский, средний)
None
>>> p.tag.involvement   # включенность говорящего в действие
None
>>> p.tag.mood          # наклонение (повелительное, изъявительное)
'indc'
>>> p.tag.number        # число (единственное, множественное)
'plur'
>>> p.tag.person        # лицо (1, 2, 3)
None
>>> p.tag.tense         # время (настоящее, прошедшее, будущее)
'past'
>>> p.tag.transitivity  # переходность (переходный, непереходный)
'intr'
>>> p.tag.voice         # залог (действительный, страдательный)
None
```

Если запрашиваемая характеристика для данного тега не определена, то возвращается None.

В написании граммем достаточно просто ошибиться; для борьбы с ошибками pymorphy2 выкидывает исключение, если встречает недопустимую граммему:

```
>>> 'foobar' in p.tag
Traceback (most recent call last):
...
ValueError: Grammeme is unknown: foobar
>>> {'NOUN', 'foo', 'bar'} in p.tag
Traceback (most recent call last):
```

```
...
ValueError: Grammemes are unknown: {'bar', 'foo'}
```

Это работает и для атрибутов:

```
>>> p.tag.POS == 'plur'
Traceback (most recent call last):
...
ValueError: 'plur' is not a valid grammeme for this attribute.
```

Склонение слов

pymorphy2 умеет склонять (ставить в какую-то другую форму) слова. Чтобы просклонять слово, его нужно сначала разобрать - понять, в какой форме оно стоит в настоящий момент и какая у него лексема:

```
>>> butyavka = morph.parse('бутявка')[0]
>>> butyavka
Parse(word='бутявка', tag=OpencorporaTag('NOUN,inan,femn sing,nomn'), normal_form='бутявка', score=1.0, methods_stack=((
```

Для склонения используйте метод Parse.inflect():

```
>>> butyavka.inflect({'gent'})  # нет кого? (родительный падеж)
Out[13]:
Parse(word='бутявки', tag=OpencorporaTag('NOUN,inan,femn sing,gent'), normal_form='бутявка', score=1.0, methods_stack=((<
>>> butyavka.inflect({'plur', 'gent'})  # кого много?
Parse(word='бутявок', tag=OpencorporaTag('NOUN,inan,femn plur,gent'), normal_form='бутявка', score=1.0, methods_stack=((<
```

С помощью атрибута Parse.lexeme можно получить лексему слова:

```
>>> butyavka.lexeme
[Parse(word='бутявка', tag=OpencorporaTag('NOUN,inan,femn sing,nomn'), normal_form='бутявка', score=1.0, methods_stack=(
 Parse(word='бутявки', tag=OpencorporaTag('NOUN,inan,femn sing,gent'), normal_form='бутявка', score=1.0, methods_stack=((
 Parse(word='бутявке', tag=OpencorporaTag('NOUN,inan,femn sing,datv'), normal_form='бутявка', score=1.0, methods_stack=((
 Parse(word='бутявку', tag=OpencorporaTag('NOUN,inan,femn sing,accs'), normal_form='бутявка', score=1.0, methods_stack=((
 Parse(word='бутявкой', tag=OpencorporaTag('NOUN,inan,femn sing,ablt'), normal_form='бутявка', score=1.0, methods_stack=((
 Parse(word='бутявкою', tag=OpencorporaTag('NOUN,inan,femn sing,ablt,V-oy'), normal_form='бутявка', score=1.0, methods_sta
 Parse(word='бутявке', tag=OpencorporaTag('NOUN,inan,femn sing,loct'), normal_form='бутявка', score=1.0, methods_stack=((<
 Parse(word='бутявки', tag=OpencorporaTag('NOUN,inan,femn plur,nomn'), normal_form='бутявка', score=1.0, methods_stack=(
 Parse(word='бутявок', tag=OpencorporaTag('NOUN,inan,femn plur,gent'), normal_form='бутявка', score=1.0, methods_stack=((
 Parse(word='бутявкам', tag=OpencorporaTag('NOUN,inan,femn plur,datv'), normal_form='бутявка', score=1.0, methods_stack=(
 Parse(word='бутявки', tag=OpencorporaTag('NOUN,inan,femn plur,accs'), normal_form='бутявка', score=1.0, methods_stack=((
 Parse(word='бутявками', tag=OpencorporaTag('NOUN,inan,femn plur,ablt'), normal_form='бутявка', score=1.0, methods_stack=
 Parse(word='бутявках', tag=OpencorporaTag('NOUN,inan,femn plur,loct'), normal_form='бутявка', score=1.0, methods_stack=((
```

Постановка слов в начальную форму

Как уже было написано, нормальную (начальную) форму слова можно получить через атрибуты Parse.normal_form и Parse.normalized.

Но что считается за нормальную форму? Например, возьмем слово "думающим". Иногда мы захотим нормализовать его в "думать", иногда - в "думающий", иногда - в "думающая".

Посмотрим, что сделает pymorphy2 в этом примере:

```
>>> m.parse('думающему')[0].normal_form
'думать'
```

pymorphy2 сейчас использует алгоритм нахождения нормальной формы, который работает наиболее быстро (берется первая форма в лексеме) - поэтому, например, все причастия сейчас нормализуются в инфинитивы. Это можно считать деталью реализации.

Если требуется нормализовывать слова иначе, можно воспользоваться методом Parse.inflect():

```
>>> m.parse('думающему')[0].inflect({'sing', 'nomn'}).word
'думающий'
```

Согласование слов с числительными

Слово нужно ставить в разные формы в зависимости от числительного, к которому оно относится. Например: “1 бутявка”, “2 бутявки”, “5 бутявок”

Для этих целей используйте метод Parse.make_agree_with_number():

```
>>> butyavka = morph.parse('бутявка')[0]
>>> butyavka.make_agree_with_number(1).word
'бутявка'
>>> butyavka.make_agree_with_number(2).word
'бутявки'
>>> butyavka.make_agree_with_number(5).word
'бутявок'
```

Выбор правильного разбора

pymorphy2 возвращает все допустимые варианты разбора, но на практике обычно нужен только один вариант, правильный.

У каждого разбора есть параметр score:

```
>>> m.parse('на')
[Parse(word='на', tag=OpencorporaTag('PREP'), normal_form='на', score=0.999628, methods_stack=((<DictionaryAnalyzer>, 'на
 Parse(word='на', tag=OpencorporaTag('INTJ'), normal_form='на', score=0.000318, methods_stack=((<DictionaryAnalyzer>, 'на'
 Parse(word='на', tag=OpencorporaTag('PRCL'), normal_form='на', score=5.3e-05, methods_stack=((<DictionaryAnalyzer>, 'на',
```

score - это оценка P(tag|word), оценка вероятности того, что данный разбор правильный.

Условная вероятность P(tag|word) оценивается на основе корпуса OpenCorpora: ищутся все неоднозначные слова со снятой неоднозначностью, для каждого слова считается, сколько раз ему был сопоставлен данный тег, и на основе этих частот вычисляется условная вероятность тега (с исползованием сглаживания Лапласа).

На данный момент оценки P(tag|word) на основе OpenCorpora есть примерно для 15 тыс. слов (исходя из примерно 110тыс. наблюдений). Для тех слов, для которых такой оценки нет, вероятность P(tag|word) либо считается равномерной (для словарных слов), либо оценивается на основе эмпирических правил (для несловарных слов).

На практике это означает, что первый разбор из тех, что возвращают методы MorphAnalyzer.parse() и MorphAnalyzer.tag(), более вероятен, чем остальные. Для слов (без учета пунктуации и т.д.) цифры такие:

- случайно выбранный разбор (из допустимых) верен примерно в 66% случаев;
- первый по словарю разбор (pymorphy2 $<$ 0.4) верен примерно в 72% случаев;
- разбор, который выдает pymorphy2 == 0.4, выбранный на основе оценки P(tag|word), верен примерно в 79% случаев.

Разборы сортируются по убыванию score, поэтому везде в примерах берется первый вариант разбора из возможных (например, morph.parse('бутявка')[0]).

Оценки P(tag|word) помогают улучшить разбор, но их недостаточно для надежного снятия неоднозначности, как минимум по следующим причинам:

- то, как нужно разбирать слово, зависит от соседних слов; pymorphy2 работает только на уровне отдельных слов;
- условная вероятность P(tag|word) оценена на основе сбалансированного набора текстов; в специализированных текстах вероятности могут быть другими - например, возможно, что в металлургических текстах P(NOUN|стали) > P(VERB|стали);
- в OpenCorpora у большинства слов неоднозначность пока не снята; выполняя задания на сайте OpenCorpora, можно непосредственно помочь улучшить оценку P(tag|word) и, следовательно, качество работы pymorphy2.

Если вы берете первый разбор из возможных (как в примерах), то стоит учитывать эту проблему.

Иногда могут помочь какие-то особенности задачи. Например, если нужно просклонять слово, и известно, что на входе ожидается слово в именительном падеже, то лучше брать вариант разбора в именительном падеже, а не первый. В общем же случае для выбора точного разбора необходимо каким-то образом учитывать не только само слово, но и другие слова в предложении.

1.1.2 Обозначения для граммем

В pymorphy2 используются словари OpenCorpora и граммемы, принятые в OpenCorpora (с небольшими изменениями).

Полный список граммем OpenCorpora доступен тут: http://opencorpora.org/dict.php?act=gram

Часть речи

Граммема	Значение	Примеры
NOUN	имя существительное	хомяк
ADJF	имя прилагательное (полное)	хороший
ADJS	имя прилагательное (краткое)	хорош
COMP	компаратив	лучше, получше, выше
VERB	глагол (личная форма)	говорю, говорит, говорил
INFN	глагол (инфинитив)	говорить, сказать
PRTF	причастие (полное)	прочитавший, прочитанная
PRTS	причастие (краткое)	прочитана
GRND	деепричастие	прочитав, рассказывая
NUMR	числительное	три, пятьдесят
ADVB	наречие	круто
NPRO	местоимение-существительное	он
PRED	предикатив	некогда
PREP	предлог	в
CONJ	союз	и
PRCL	частица	бы, же, лишь
INTJ	междометие	ой

Часть речи можно получить через атрибут POS:

```
>>> p = morph.parse('идти')[0]
>>> p.tag.POS
'INFN'
```

Падеж

Граммема	Значение	Пояснение	Примеры
nomn	именительный	Кто? Что?	хомяк ест
gent	родительный	Кого? Чего?	у нас нет хомяка
datv	дательный	Кому? Чему?	сказать хомяку спасибо
accs	винительный	Кого? Что?	хомяк читает книгу
ablt	творительный	Кем? Чем?	зерно съедено хомяком
loct	предложный	О ком? О чём? и т.п.	хомяка несут в корзинке
voct	звательный	Его формы используются при обращении к человеку.	Саш, пойдем в кино.
gen2	второй родительный (частичный)		ложка сахару (gent - производство сахара); стакан яду (gent - нет яда)
acc2	второй винительный		записался в солдаты
loc2	второй предложный (местный)		я у него в долгу (loct - напоминать о долге); висит в шкафу (loct - монолог о шкафе); весь в снегу (loct - писать о снеге)

Падеж выделяется у существительных, полных прилагательных, полных причастий, числительных и местоимений. Получить его можно через атрибут case:

```
>>> p = morph.parse('хомяку')[0]
>>> p.tag.case
'datv'
```

Примечание: В OpenCorpora (на июль 2013) есть еще падежи gen1 и loc1. Они указываются вместо gent/loct, когда у слова есть форма gen2/loc2. В pymorphy2 gen1 и loc1 заменены на gent/loct, чтоб с ними было проще работать.

Число

Граммема	Значение	Примеры
sing	единственное число	хомяк, говорит
plur	множественное число	хомяки, говорят

```
>>> p = morph.parse('люди')[0]
>>> p.tag.number
'plur'
```

Нестандартные граммемы

В pymorphy2 используются некоторые граммемы, отсутствующие в словаре OpenCorpora:

Граммема	Значение
LATN	Токен состоит из латинских букв (например, "foo-bar" или "Maßstab")
PNCT	Пунктуация (например, , или !? или ...)
NUMB	Число (например, "204")
ROMN	Римское число (например, XI)

Пример:

```
>>> p = morph.parse('...')[0]
>>> p.tag
OpencorporaTag('PNCT')
```

1.1.3 Как принять участие в разработке

Общая информация

Исходный код pymorphy2 распространяется по лицензии MIT и доступен на github и bitbucket:

- https://github.com/kmike/pymorphy2
- https://bitbucket.org/kmike/pymorphy2

Баг-трекер - на гитхабе. Для общения можно использовать гугл-группу (есть какие-то идеи, предложения, замечания - пишите).

Если вы хотите улучшить код pymorphy2 - может быть полезным ознакомиться с разделом Внутреннее устройство.

pymorphy2 работает под Python 2.x и 3.x без использования утилиты 2to3; написание такого кода, по опыту, оказывается не сложнее написания кода просто под 2.x, но поначалу требует некоторой внимательности и осторожности. Пожалуйста, пишите и запускайте тесты, если что-то меняете.

Улучшать можно не только код - улучшения в документации, идеи и сообщения об ошибках тоже очень ценны.

pymorphy2 основывается на словарях из OpenCorpora и использует наборы текстов оттуда для автоматического тестирования и замеров скорости; в будущем планируется также использовать размеченный корпус для снятия неоднозначности разбора, ну и вцелом это классный проект. Любая помощь OpenCorpora - это вклад и в pymorphy2.

Тестирование

Запуск

Тесты лежат в папке tests. При написании тестов используется pytest. Для их запуска используется утилита tox, которая позволяет выполнять тесты для нескольких интерпретаторов питона.

Для запуска тестов установите tox через pip:

```
pip install tox
```

и выполните

```
tox
```

из папки с исходным кодом.

Замеры скорости работы

Чтобы улучшать скорость работы pymorphy2 (оптимизируя структуры данных, алгоритмы или реализацию), нужны какие-то надежные средства для замеров производительность.

На каких данных измерять?

Самый очевидный способ - выполнить интересующую операцию pymorphy2 для всех слов из какого-то большого текста или корпуса. Это полезная метрика, которая позволяет оценить среднюю скорость разбора. Минусы - зависимость от конкретного корпуса, необходимость этот корпус распространять.

В pymorphy2 для замеров этого типа вместо корпуса целиком используются частотные списки униграмм из OpenCorpora: для каждого слова разбор можно выполняеть столько раз, сколько оно встречалось в корпусе.

Для того, чтоб бороться с "эффектом хоббита", выполняется еще одна проверка производительности - разбор прогоняется по 1 разу для каждого слова из top-100k униграмм.

Для оценки производительности поддержки буквы "ё" во всех словах "ё" заменяется на "е" и после этого опять прогоняется бенчмарк.

Организация замеров производительности в pymorphy2

Код для бенчмарков лежит в папке benchmarks. Для запуска тестов производительности выполните

```
tox -c bench.ini
```

из папки с исходным кодом pymorphy2.

Замеры потребления памяти

Чтобы оптимизировать структуру словаря, нужно отслеживать потребление памяти. Для этого у утилиты командной строки pymorphy есть специальная команда:

```
pymorphy dict mem_usage <FILE> [--verbose]
```

Для ее работы нужно предварительно установить psutil:

```
pip install psutil
```

<FILE> - это папка со словарем; если указана опция --verbose, будет напечатан более детальный отчет о том, объекты каких типов занимают сколько памяти.

Примечание: Для работы --verbose требуется установить пакет guppy, который на данный момент недоступен для python 3 (+ для работы под 2.7 трубеутся устанавливать последнюю версию guppy из svn).

1.2 Внутреннее устройство

1.2.1 Словари

В pymorphy2 используются словари из проекта OpenCorpora, специальным образом обработанные для быстрых выборок.

Упаковка словаря

Исходный словарь из OpenCorpora представляет собой файл, в котором слова объединены в лексемы следующим образом:

```
1
ёж      NOUN,anim,masc sing,nomn
ежа     NOUN,anim,masc sing,gent
ежу     NOUN,anim,masc sing,datv
ежа     NOUN,anim,masc sing,accs
ежом    NOUN,anim,masc sing,ablt
еже     NOUN,anim,masc sing,loct
ежи     NOUN,anim,masc plur,nomn
ежей    NOUN,anim,masc plur,gent
ежам    NOUN,anim,masc plur,datv
ежей    NOUN,anim,masc plur,accs
ежами   NOUN,anim,masc plur,ablt
ежах    NOUN,anim,masc plur,loct
```

Сначала указывается номер лексемы, затем перечисляются формы слова и соответствующая им грамматическая информация (тег). Первой формой в списке идет нормальная форма слова. В словаре около 400тыс. лексем и 5млн отдельных слов.

Примечание: С сайта OpenCorpora для скачивания доступны plaintext- и XML-версии словаря. В pymorphy2 используется XML-версия, но (для простоты) тут и далее в примерах показан plaintext-формат. Сами данные в XML-версии те же.

Если просто загрузить все слова и их грамматическую информацию в питоний list, то это займет примерно 2Гб оперативной памяти. Кроме того, эта форма неудобна для быстрого выполнения операций по анализу и склонению слов.

Упаковка грамматической информации

Каждым тегом (например, NOUN,anim,masc sing,nomn) обычно помечено более одного слова (часто - очень много слов). Хранить строку целиком для всех 5млн слов накладно по 2 причинам:

- в питоне не гарантировано, что id(string1) == id(string2), если string1 == string2 (хотя функция intern может помочь);
- строки нельзя хранить в array.array, а у list накладные расходы выше, т.к. он в питоне реализован как массив указателей на объекты, поэтому в случае с тегами важно, чтоб каждому слову была сопоставлена цифра, а не строка.

В pymorphy2 все возможные теги хранятся в массиве (в list); для каждого слова указывается только номер тега.

Пример:

```
1
ёж      1
ежа     2
ежу     3
ежа     4
ежом    5
еже     6
ежи     7
ежей    8
ежам    9
ежей    10
ежами   11
ежах    12
```

набор тегов:

```
['NOUN,anim,masc sing,nomn',
 'NOUN,anim,masc sing,gent',
 'NOUN,anim,masc sing,datv',
 'NOUN,anim,masc sing,accs',
 'NOUN,anim,masc sing,ablt',
 'NOUN,anim,masc sing,loct',
 'NOUN,anim,masc plur,nomn',
 'NOUN,anim,masc plur,gent',
 'NOUN,anim,masc plur,datv',
 'NOUN,anim,masc plur,accs',
 'NOUN,anim,masc plur,ablt',
 'NOUN,anim,masc plur,loct',
 # ...
]
```

Парадигмы

Изначально в словаре из OpenCorpora нет понятия парадигмы слова (парадигма - это образец для склонени или спряжения слов). В pymorphy2 выделенные явным образом словоизменительные парадигмы необходимы для того, чтоб склонять неизвестные слова (т.к. при этом нужны образцы для склонения).

Примечание: Для других операций явно выделенные парадигмы тоже могут быть удобными, хотя все, кроме склонения неизвестных слов, можно было бы выполнять достаточно быстро и без явно выделенных парадигм.

Пример исходной лексемы:

```
тихий   100
тихого  102
тихому  105
...
тише    124
потише  148
```

У слов в этой лексеме есть неизменяемая часть (стем "ти"), изменяемое "окончание" и необязательный "префикс" ("по"). Выделив у каждой формы "окончание" и "префикс", можно разделить лексему на стем и таблицу для склонения:

```
стем: ти
таблица для склонения ("окончание", номер тега, "префикс"):

    "хий"     100  ""
    "хого"    102  ""
    "хому"    105  ""
    ...
    "ше"      124  ""
    "ше"      125  "по"
```

Для многих лексем таблицы для склонения получаются одинаковыми. В pymorphy2 выделенные таким образом таблицы для склонения принимаются за парадигмы.

"Окончания" и "префиксы" в парадигмах повторяются, и хорошо бы их не хранить по многу раз (а еще лучше - создавать поменьше питоньих объектов для них), поэтому все возможные "окончания" хранятся в отдельном массиве, а в парадигме указывается только номер "окончания"; с "префиксами" - то же самое.

В итоге получается примерно так:

```
55      100     0
56      102     0
57      105     0
...
73      124     0
73      125     1
```

Примечание: Сейчас все возможные окончания парадигм хранятся в list; возможно, было бы более эффективно хранить их в DAWG или Trie и использовать perfect hash для сопоставления индекс <-> слово, но сейчас это не реализовано.

Линеаризация парадигм

Тройки "окончание, номер грамматической информации, префикс" в tuple хранить расточительно, т.к. этих троек получается очень много (сотни тысяч), а каждый tuple требует дополнительной памяти:

```
>>> import sys
>>> sys.getsizeof(tuple())
56
```

Поэтому каждая парадигма упаковывается в одномерный массив: сначала идут все номера окончаний, потом все номера тегов, потом все номера префиксов:

```
55 56 57 ... 73 73 | 100 102 105 ... 124 125 | 0 0 0 ... 0 1
```

Пусть парадигма состоит из N форм слов; в массиве будет тогда N*3 элементов. Данные о i-й форме можно получить с помощью индексной арифметики: например, номер грамматической информации для формы с индексом 2 (индексация с 0) будет лежать в элементе массива с номером N + 2, а номер префикса для этой же формы - в элементе N*2 + 2.

Хранить числа в питоньем list накладно, т.к. числа типа int - это тоже объекты и требуют памяти:

```
>>> import sys
>>> sys.getsizeof(1001)
24
```

Память под числа [-5...256] в CPython выделена заранее, но

- это деталь реализации CPython;
- в парадигмах много чисел не из этого интервала;
- list в питоне реализован через массив указателей, а значит требует дополнительные 4 или 8 байт на элемент (на 32- и 64-битных системах).

Поэтому данные хранятся в array.array из стандартной библиотеки.

Связи между лексемами

В словаре OpenCorpora доступна информация о связях между лексемами. Например, может быть связана лексема для инфинитива и лексема с формами глагола, соответствующими этому инфинитиву. Или, например, формы краткого и полного прилагательного.

Эта информация позволяет склонять слова между частями речи (например, причастие приводить к глаголу).

В pymorphy2 все связанные лексемы просто объединяются в одну большую лексему на этапе подготовки (компиляции) исходного словаря; в скомпилированном словаре информация о связях между лексемами в явном виде недоступна.

Упаковка слов

Для хранения данных о словах используется граф (Directed Acyclic Word Graph, wiki) с использованием библиотек DAWG (это обертка над C++ библиотекой dawgdic) или DAWG-Python (это написанная на питоне реализация DAWG, которая не требует компилятора для установки и работает быстрее DAWG под PyPy).

В структуре данных DAWG некоторые общие части слов не дублируются (=> требуется меньше памяти); кроме того, в DAWG можно быстро выполнять не только точный поиск слова, но и другие операции - например, поиск по префиксу или поиск с заменами.

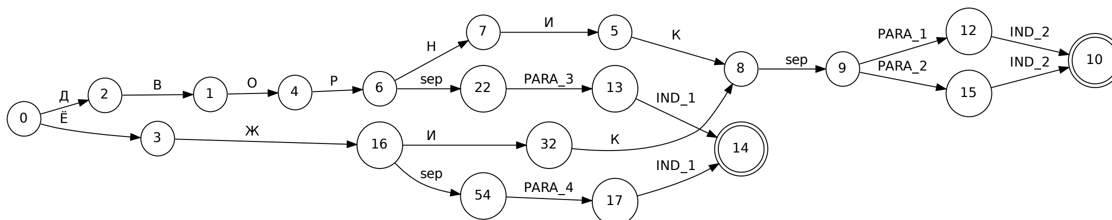
В pymorphy2 в DAWG помещаются не сами слова, а строки вида

> <слово> <разделитель> <номер парадигмы> <номер формы в парадигме>

Пусть, для примера, у нас есть слова (в скобках - допустимые разборы, определяемые парами "номер парадигмы, номер формы в парадигме").

```
двор     (3, 1)
ёж       (4, 1)
дворник (1, 2) и (2, 2)
ёжик    (1, 2) и (2, 2)
```

Тогда они будут закодированы в такой граф:



Этот подход позволяет экономить память (т.к. как сами слова, так и данные о парадигмах и индексах сжимаются в DAWG), + алгоритмы упрощаются: например, для получения всех возможных вариантов разбора слова достаточно найти все ключи, начинающиеся с

<слово> <разделитель>

– а эта операция (поиск всех ключей по префиксу) в используемой реализации DAWG достаточно эффективная. Хранение слов в DAWG позволяет также быстро и правильно обрабатывать букву "ё".

Примечание: На самом деле граф будет немного не такой, т.к. текст кодируется в utf-8, а значения в base64, и поэтому узлов будет больше; для получения одной буквы или цифры может требоваться совершить несколько переходов.

Кодировка utf-8 используется из-за того, что кодек utf-8 в питоне в несколько раз быстрее однобайтового cp1251. Кодировка цифр в base64 - тоже деталь реализации: C++ библиотека, на которой основан DAWG, поддерживает только нуль-терминированные строки. Байт 0 считается завершением строки и не может присутствовать в ключе, а для двухбайтовых целых числел сложно гарантировать, что оба байта ненулевые.

Примечание: Подход похож на тот, что описан на aot.ru.

Итоговый формат данных

Таблица с грамматической информацией

```
['tag1', 'tag2', ...]
```

tag<N> - тег (грамматическая информация, набор граммем): например, NOUN,anim,masc sing,nomn.

Этот массив занимает где-то 0.5M памяти.

Парадигмы

```
paradigms = [
    array.array("<H", [
        suff_id1, .., suff_idN,
        tag_id1, .., tag_idN,
        pref_id1, .., pref_idN
    ]),

    array.array("<H", [
        ...
    ]),

    ...
]

suffixes = ['suffix1', 'suffix2', ...]
prefixes = ['prefix1', 'prefix2', ...]
```

suff_id<N>, tag_id<N> и pref_id<N> - это индексы в таблицах с возможными "окончаниями" suffixes, грамматической информацией (тегами) и "префисками" prefixes соответственно.

Парадигмы и соответствующие списки "окончаний" и "префиксов" занимают примерно 3-4M памяти.

Слова

Все слова хранятся в dawg.RecordDAWG:

```
dawg.RecordDAWG

    'word1': (para_id1, para_index1),
    'word1': (para_id2, para_index2),
    'word2': (para_id1, para_index1),
    ...
```

В DAWG эта информация занимает примерно 7M памяти.

Алгоритм разбора по словарю

С описанной выше структурой словаря разбирать известные слова достаточно просто. Код на питоне:

```python
result = []

# Ищем в DAWG со словами все ключи, которые начинаются
# с <СЛОВО><sep> (обходом по графу); из этих ключей (из того, что за <sep>)
# получаем список кортежей [(para_id1, index1), (para_id2, index2), ...].
#
# RecordDAWG из библиотек DAWG или DAWG-Python умеет это делать
# одной командой (с возможностью нечеткого поиска для буквы Ё):

para_data = self._dictionary.words.similar_items(word, self._ee)

# fixed_word - это слово с исправленной буквой Ё, для которого был
# проведен разбор.

for fixed_word, parse in para_data:
    for para_id, idx in parse:

        # по информации о номере парадигмы и номере слова в
        # парадигме восстанавливаем нормальную форму слова и
        # грамматическую информацию.

        tag = self._build_tag_info(para_id, idx)
        normal_form = self._build_normal_form(para_id, idx, fixed_word)

        result.append(
            (fixed_word, tag, normal_form)
        )
```

Настоящий код немного отличается в деталях, но суть та же.

Т.к. парадигмы запакованы в линейный массив, требуются дополнительные шаги для получения данных. Метод _build_tag_info реализован, например, вот так:

```python
def _build_tag_info(self, para_id, idx):

    # получаем массив с данными парадигмы
    paradigm = self._dictionary.paradigms[para_id]

    # индексы грамматической информации начинаются со второй трети
    # массива с парадигмой
    tag_info_offset = len(paradigm) // 3
```

```
    # получаем искомый индекс
    tag_id = paradigm[tag_info_offset + tag_id_index]

    # возвращаем соответствующую строку из таблицы с грамматической информацией
    return self._dictionary.gramtab[tag_id]
```

Примечание: Для разбора слов, которых нет в словаре, в pymorphy2 есть предсказатель.

Формат хранения словаря

Итоговый словарь представляет собой папку с файлами:

```
dict/
    meta.json
    gramtab-opencorpora-int.json
    gramtab-opencorpora-ext.json
    grammemes.json
    suffixes.json
    paradigm-prefixes.json
    paradigms.array
    words.dawg
    prediction-suffixes-0.dawg
    prediction-suffixes-1.dawg
    prediction-suffixes-2.dawg
    prediction-prefixes.dawg
```

Файлы .json - обычные json-данные; .dawg - это двоичный формат C++ библиотеки dawgdic; paradigms.array - это массив чисел в двоичном виде.

Примечание: Если вы вдруг пишете морфологический анализатор не на питоне (и формат хранения данных устраивает), то вполне возможно, что будет проще использовать эти подготовленные словари, а не конвертировать словари из OpenCorpora еще раз; ничего специфичного для питона в сконвертированных словарях нет.

Характеристики

После применения описанных выше методов в pymorphy2 словарь со всеми сопутствующими данными занимает около 15Мб оперативной памяти; скорость разбора - от нескольких десятков тыс. слов/сек до > 100тыс. слов/сек (в зависимости от интерпретатора, настроек и выполняемой операции). Для сравнения:

- в mystem словарь + код занимает около 20Мб оперативной памяти, скорость > 100тыс. слов/сек;
- в lemmatizer из aot.ru словарь занимает 9Мб памяти (судя по данным отсюда), скорость > 200тыс слов/сек.;
- в варианте морф. анализатора на конечных автоматах с питоновской оберткой к openfst (http://habrahabr.ru/post/109736/) сообщается, что словарь занимал 35/3 = 11Мб после сжатия, скорость порядка 2 тыс слов/сек без оптимизаций;
- написанный на питоне вариант морф. анализатора на конечных автоматах (автор - Konstantin Selivanov) требовал порядка 300Мб памяти, скорость порядка 2 тыс. слов/сек;

- в pymorphy 0.5.6 полностью загруженный в память словарь (этот вариант там не документирован) занимает порядка 300Мб, скорость порядка 1-2тыс слов/сек.
- Про MAnalyzer v0.1 (основанный на алгоритмах из pymorphy1, но написанный на C++ и с использованием dawg) приводят сведения, что скорость разбора 900тыс слов/сек при потреблении памяти 40Мб;
- в первом варианте формата словарей pymorphy2 (от которого я отказался) получалась скорость 20-60тыс слов/сек при 30М памяти или 2-5 тыс слов/сек при 5Мб памяти (предсказатель там не был реализован).

Цели обогнать C/C++ реализации у pymorphy2 нет; цель - скорость базового разбора должна быть достаточной для того, чтоб "продвинутые" операции работали быстро. Мне кажется, 30 тыс. слов/сек или 300 тыс. слов/сек - это не очень важно для многих задач, т.к. накладные расходы на обработку и применение результатов разбора все равно, скорее всего, "съедят" эту разницу (особенно при использовании из питоньего кода).

Предупреждение: Информация в этом разделе немного устарела.

1.2.2 Предсказатель

В тех случаях, когда слово не получается найти простым поиском по словарю (с учетом буквы "ё"), в дело вступает предсказатель.

Предсказатель основан на идее о том, что если слова в русском языке оканчиваются одинаково, то и форму они имеют, скорее всего, одинаковую.

Примечание: Алгоритм предсказания в основе своей похож на тот, что описан на aot.ru, и на тот, что применяется в pymorphy1, но отличается в деталях и содержит дополнительные эвристики.

Для предсказателя реализованы 2 алгоритма предсказания, которые работают совместно.

Первый подход: отсечение префиксов

Если 2 слова отличаются только тем, что к одному из них что-то приписано спереди, то, скорее всего, и склоняться они будут одинаково.

Это особенно справедливо в тех случаях, когда это "что-то" - один из известных словообразовательных префиксов (например, "кошка" - "псевдокошка").

В pymorphy2 хранится небольшой список таких префиксов (например, "не", "анти", "псевдо", "супер", "дву" и т.д.); если слово начинается с одного из таких префиксов, то префикс отсекается, а остаток передается на разбор.

Если слово не начинается с такого префикса, то анализатор все равно пробует разобрать слово путем отсечения префикса: сначала он пробует считать одну первую букву слова префиксом, потом 2 первых буквы и т.д., и пытается при этом анализировать то, что осталось (это делается только для не очень длинных префиксов и не очень коротких остатков).

Второй подход: предсказание по концу слова

В подходе с отсечением префиксов есть два принципиальных ограничения:

- разбор не должен зависеть от префикса (что неверно для словоизменительных префиксов "по" и "наи", которые образуют формы прилагательных);

- морфологический анализатор должен уметь разбирать правую часть слова (путем поиска по словарю или еще как-то) - правая часть слова должна иметь какой-то смысл сама по себе.

Разбор многих слов нельзя предсказать, отсекая префикс и разбирая остаток. Например, хотелось бы, чтоб если в словаре было слово "кошка", но не было "мошка" и "ошка", на основе словарного слова "кошка" анализатор смог бы предположить, как склоняется "мошка" (т.к. они заканчиваются одинаково).

Для того, чтоб предсказывать формы слов по тому, как слова заканчиваются, при конвертации словарей строится DAWG со всеми возможными окончаниями слов (в данный момент - от однобуквенных до пятибуквенных); каждому окончанию сопоставляется массив с возможными вариантами разбора слов с такими окончаниями.

Схема хранения похожа на ту, что в основном словаре (см. раздел Упаковка слов), только

- вместо самих слов хранятся все их возможные окончания;
- к номеру парадигмы и индексу формы в парадигме добавляется еще "продуктивность" данного правила - количество слов в словаре, которые имеют данное окончание и разбираются данным образом.

```
<конец слова> <разделитель> <продуктивность> <номер парадигмы> <номер формы в парадигме>
```

Если для каждого "окончания" хранить все возможные варианты разбора, то получится заведомо много лишних (очень маловероятных) правил. Поэтому скрипт компиляции словаря умеет отсекать правила по нескольким критериям:

- парадигма должна быть "продуктивной": в словаре должно иметься хотя бы min_paradigm_popularity лемм, разбираемых по этой парадигме;
- "окончания" должны быть распространенными: в словаре должно иметься хотя бы min_ending_freq слов, которые заканчиваются так;
- вариант разбора должен быть популярным: для данного окончания для каждой части речи оставляем только самые популярные варианты разбора;

По умолчанию min_paradigm_popularity == 3, min_ending_freq == 2.

Разбор сводится к поиску наиболее длинной правой части разбираемого слова, которая есть в DAWG с окончаниями.

Кроме того, для каждого словоизменительного префикса (ПО, НАИ) точно так же строится еще по одному DAWG; если слово начинается с одного из этих префиксов, то анализатор добавляет к результату варианты предсказания, полученные поиском по соответствующему DAWG.

Примечание: Термин "окончание" тут употребляется в смысле "правая часть слова определенной длины"; он не имеет отношения к "школьному" определению; кроме того, тут он не имеет отношения к "окончаниям" в парадигмах.

Ограничение на части речи

В русском языке не все части речи продуктивные: например, нельзя приписать что-то к предлогу, чтоб получить другой предлог; все предлоги есть в словаре, и предсказывать незнакомые слова как предлоги неправильно. Такие варианты предсказания отбрасываются предсказателем.

Слова, записанные через дефис

Предупреждение: Разбор слов, записанных через дефис, еще не реализован.

Сортировка результатов разбора

При предсказании по концу слова результаты сортируются по "продуктивности" вариантов разбора: наиболее продуктивные варианты будут первыми.

Другими словами, варианты разбора (= номера парадигм) упорядочены по частоте, с которой эти номера парадигм соответствуют данному окончанию для данной части речи - без учета частотности по корпусу.

Экспериментального подтверждения правильности этого подхода нет, но "интуиция" тут такая:

1. нам не важно, какие слова в корпусе встречаются часто, т.к. предсказатель работает для редких слов, и редкие слова он должен предсказывать как редкие, а не как распространенные;
2. для "длинного хвоста" частотности в корпусе конкретные цифры имеют не очень много значения, т.к. флуктуации очень большие, "эффект хоббита" и т.д.
3. С другой стороны, важно, какие парадигмы в русском языке более продуктивные, какие порождают больше слов.

Поэтому используется частотность по парадигмам, полученная исключительно из словаря.

Примечание: В настоящий момент результаты сортируются только при предсказании по концу слова. Разборы для словарных слов и разборы, предсказанные путем отсечения префикса, специальным образом сейчас не сортируются.

Оценки для вариантов разбора

pymorphy2 приписывает каждому варианту разбора число (0.0 < x <= 1.0); это число может служить оценкой того, насколько анализатор уверен в данном варианте разбора.

Например, оценка 1.0 означает, что слово найдено в словаре, а оценка 0.001 будет свидетельствовать о том, что это редкий вариант разбора, предложенный предсказателем.

Предупреждение: Это очень экспериментальная возможность.
Оценки не стоит рассматривать как значения вероятностей правильности разбора. Более того, никаких подтверждений связи вероятности правильности разбора с оценкой предсказателя у меня тоже нет; "коэффициенты", на основе которых вычисляются оценки, выбраны вручную достаточно произвольно.

1.2.3 Буква Ё

Если не ударяться в крайности, то можно считать, что в русском языке употребление буквы "ё" допустимо, но не обязательно. Это означает, что как в исходном тексте, так и в словарях она иногда может быть, а иногда ее может не быть.

В pymorphy2 считается, что:

- в словарях употребление буквы "ё" обязательно; "е" вместо "ё" (как и "ё" вместо "е") - это ошибка в словаре. Иными словами, "е" и "ё" в словарях - две совсем разные буквы.
- В текстах/словах, которые подаются на вход морфологического анализатора, употребление буквы "ё" необязательно. Например, слово "озера" должно быть разобрано и как "(нет) озера", и как "(глубокие) озёра".

Примечание: При этом входное слово "озёра" будет однозначно разобрано как "(глубокие) озёра".

Детали реализации

"Наивный" подход - это генерация все вариантов возможных замен "е" на "ё" во входном слове и проверка всех вариантов по словарю. В русском языке "е" - очень распространенная буква, и много слов, где "е" встречается несколько раз. Например, для слова с 3 буквами "е" нужно сгенерировать еще 7 вариантов слова - вместо 1 проверки по словарю нужно было бы выполнить 8 (+ время на генерацию вариантов слов).

При разборе pymorphy2 использует другой подход - все слова хранятся в графе, и при обходе графа кроме направлений "е" каждый раз еще пробуется направление "ё". При этом в исходном коде pymorphy2 этого обхода графа в явном виде нет, т.к. библиотеки DAWG и DAWG-Python сами умеет производить "поиск с возможными заменами".

"Наивный" подход в pymorphy2 используется только в скрипте генерации данных для автоматических тестов (чтобы обеспечить перекрестную проверку).

Примечание: По оценкам, полученным в начале разработки (которые, соответственно, могут быть неверными для текущей версии), поддержка буквы "ё" в pymorphy2 замедляла разбор на 10-40% (в зависимости от интерпретатора).

1.3 Разное

1.3.1 История изменений

0.4 (2013-10-19)

- Parse.estimate переименован в score и содержит теперь оценку P(tag|word) на основе данных из OpenCorpora;
- по умолчанию результаты разбора сортируются по score.

То, что результатам сопоставляется оценка P(tag|word), может в некоторых случаях снизить скорость разбора раза в 1.5 - 2. Если эти оценки не нужны, создайте экземпляр MorphAnalyzer с параметром probability_estimator_cls=None.

Для обновления требуется обновить pymorphy2-dicts до версии >= 2.4, а также библиотеки DAWG или DAWG-Python до версий >= 0.7.

0.3.5 (2013-06-30)

- Препроцессинг словаря: loc1/gen1/acc1 заменяются на loct/gent/accs; варианты написания тегов унифицируются (чтоб их было меньше);

- исправлено согласование слов с числительными;
- при склонении слов в loc2/gen2/acc2/voct слово ставится в loct/gent/accs/nomn, если вариантов с loc2/gen2/acc2/voct не найдено.

Для полноценного обновления лучше обновить pymorphy2-dicts до версии >= 2.2.

0.3.4 (2013-04-29)

- Добавлен метод Parse.make_agree_with_number для согласования слов с числительными;
- небольшие улучшения в документации.

0.3.3 (2013-04-12)

- Исправлен тег, который выдает RomanNumberAnalyzer (теперь это ROMN, как в OpenCorpora);
- добавлена функция pymorphy2.tokenizers.simple_word_tokenize, которая разбивает текст по пробелам и пунктуации (но не дефису);
- исправлена ошибка с разбором слов вроде "ретро-fm" (pymorphy2 раньше падал с исключением).

0.3.2 (2013-04-03)

- добавлен RomanNumberAnalyzer для разбора римских чисел;
- MorphAnalyzer и OpencorporaTag теперь можно сериализовывать с помощью pickle;
- улучшены тесты;
- при компиляции словаря версия xml печатается раньше.

0.3.1 (2013-03-12)

- Поправлен метод MorphAnalyzer.word_is_known, который раньше учитывал регистр слова (что неправильно);
- исправлена ошибка в разборе слов с дефисом (тех, у которых лишний дефис справа или слева).

0.3 (2013-03-11)

- Рефакторинг: теперь при необходимости можно дописывать свои "шаги" морфологического анализа ("предсказатели") и комбинировать их с существующими (документация пока не готова, и API может поменяться);
- на вход больше не обязательно подавать слова в нижнем регистре (но на выходе при этом регистр сохраняться не обязан - используйте функцию pymorphy2.shapes.restore_word_case, если требуется восстановить регистр полученных слов);
- улучшено предсказание неизвестных слов по словообразовательным префиксам (учитывается больше таких префиксов);
- реализован разбор (и склонение) слов с дефисами;
- результаты разбора теперь включают в себя полную информацию о том, как слово разбиралось; наличие para_id и idx при этом больше не обязательно;

- анализатор теперь отмечает пунктуацию тегом PNCT, числа - тегом NUMB, слова, записанные латиницей - тегом LATN;
- улучшено предсказание по неизвестному префиксу (добавлено ограничение по граммеме Apro);
- улучшения в тестах и бенчмарках;
- удален атрибут morph.dict_meta (используйте morph.dictionary.meta);
- удален (возможно, временно) метод MorphAnalyzer.inflect (используйте метод inflect у результата разбора);
- удален метод MorphAnalyzer.decline (используйте parse.lexeme);
- удалено свойство Parse.paradigm.

В результате этих изменений улучшилось качество разбора, качество склонения и возможности по расширению библиотеки (втч для настройки под конкретную задачу), но скорость работы "из коробки" по сравнению с 0.2 снизилась примерно на треть.

0.2 (2013-02-18)

- Улучшения в предсказателе: учет словоизменительных префиксов;
- улучшения в предсказателе: равноценные варианты разбора не отбрасываются;
- изменена схема проверки совместимости словарей;
- изменен формат словарей (нужно обновить pymorphy2-dicts до 2.0);
- добавлено свойство Parse.paradigm.

0.1 (2013-02-14)

Первый альфа-релиз. Релизована основа: эффективный разбор и склонение, обновление словарей, полная поддержка буквы ё.

Многие вещи, которые были доступны в pymorphy, пока не работают (разбор слов с дефисом, разбор фамилий, поддержка шаблонов django, утилиты из contrib).

Кроме того, API пока не зафиксирован и может меняться в последующих релизах.

1.3.2 Authors and Contributors

- Mikhail Korobov;
- @radixvinni;
- @ivirabyan.

If you contributed to pymorphy2, please add yourself to this list (or update your contact information).

Many people contributed to pymorphy2 predecessor, pymorphy; they are listed here: https://github.com/kmike/pymorphy/blob/master/AUTHORS.rst

1.3.3 API Reference (auto-generated)

Morphological Analyzer

class pymorphy2.analyzer.DummySingleTagProbabilityEstimator(dict_path)

apply_to_parses(word, word_lower, parses)

apply_to_tags(word, word_lower, tags)

class pymorphy2.analyzer.MorphAnalyzer(path=None, result_type=<class 'pymorphy2.analyzer.Parse'>, units=None, probability_estimator_cls=<class 'pymorphy2.analyzer.SingleTagProbabilityEstimator'>)

Morphological analyzer for Russian language.

For a given word it can find all possible inflectional paradigms and thus compute all possible tags and normal forms.

Analyzer uses morphological word features and a lexicon (dictionary compiled from XML available at OpenCorpora.org); for unknown words heuristic algorithm is used.

Create a MorphAnalyzer object:

```
>>> import pymorphy2
>>> morph = pymorphy2.MorphAnalyzer()
```

MorphAnalyzer uses dictionaries from pymorphy2-dicts package (which can be installed via pip install pymorphy2-dicts).

Alternatively (e.g. if you have your own precompiled dictionaries), either create PYMORPHY2_DICT_PATH environment variable with a path to dictionaries, or pass path argument to pymorphy2.MorphAnalyzer constructor:

```
>>> morph = pymorphy2.MorphAnalyzer('/path/to/dictionaries')
```

By default, methods of this class return parsing results as namedtuples Parse. This has performance implications under CPython, so if you need maximum speed then pass result_type=None to make analyzer return plain unwrapped tuples:

```
>>> morph = pymorphy2.MorphAnalyzer(result_type=None)
```

DEFAULT_UNITS = [<class 'pymorphy2.units.by_lookup.DictionaryAnalyzer'>, <class 'pymorphy2.units.by_shape

ENV_VARIABLE = u'PYMORPHY2_DICT_PATH'

TagClass

Тип результата pymorphy2.tagset.OpencorporaTag

classmethod choose_dictionary_path(path=None)

get_lexeme(form)
Return the lexeme this parse belongs to.

iter_known_word_parses(prefix=u'')
Return an iterator over parses of dictionary words that starts with a given prefix (default empty prefix means "all words").

normal_forms(word)
Return a list of word normal forms.

parse(word)
Analyze the word and return a list of pymorphy2.analyzer.Parse namedtuples:

Parse(word, tag, normal_form, para_id, idx, _score)

(or plain tuples if result_type=None was used in constructor).

tag(word)

word_is_known(word, strict_ee=False)
Check if a word is in the dictionary. Pass strict_ee=True if word is guaranteed to have correct e/ё letters.

Примечание: Dictionary words are not always correct words; the dictionary also contains incorrect forms which are commonly used. So for spellchecking tasks this method should be used with extra care.

---

class pymorphy2.analyzer.Parse
Parse result wrapper.

inflect(required_grammemes)

is_known
True if this form is a known dictionary form.

lexeme
A lexeme this form belongs to.

make_agree_with_number(num)
Inflect the word so that it agrees with num

normalized
A Parse instance for self.normal_form.

class pymorphy2.analyzer.SingleTagProbabilityEstimator(dict_path)

apply_to_parses(word, word_lower, parses)

apply_to_tags(word, word_lower, tags)

Analyzer units

Dictionary analyzer unit

class pymorphy2.units.by_lookup.DictionaryAnalyzer(morph)
Analyzer unit that analyzes word using dictionary.

get_lexeme(form)
Return a lexeme (given a parsed word).

parse(word, word_lower, seen_parses)
Parse a word using this dictionary.

tag(word, word_lower, seen_tags)
Tag a word using this dictionary.

Analogy analyzer units This module provides analyzer units that analyzes unknown words by looking at how similar known words are analyzed.

class pymorphy2.units.by_analogy.KnownPrefixAnalyzer(morph)
Parse the word by checking if it starts with a known prefix and parsing the reminder.

Example: псевдокошка -> (псевдо) + кошка.

class pymorphy2.units.by_analogy.KnownSuffixAnalyzer(morph)
Parse the word by checking how the words with similar suffixes are parsed.

Example: бутявкать -> ...вкать

class FakeDictionary(morph)
This is just a DictionaryAnalyzer with different __repr__

class pymorphy2.units.by_analogy.UnknownPrefixAnalyzer(morph)
Parse the word by parsing only the word suffix (with restrictions on prefix & suffix lengths).

Example: байткод -> (байт) + код

Analyzer units for unknown words with hyphens

class pymorphy2.units.by_hyphen.HyphenAdverbAnalyzer(morph)
Detect adverbs that starts with "по-".

Example: по-западному

class pymorphy2.units.by_hyphen.HyphenSeparatedParticleAnalyzer(morph)
Parse the word by analyzing it without a particle after a hyphen.

Example: смотри-ка -> смотри + "-ка".

Примечание: This analyzer doesn't remove particles from the result so for normalization you may need to handle particles at tokenization level.

class pymorphy2.units.by_hyphen.HyphenatedWordsAnalyzer(morph)
Parse the word by parsing its hyphen-separated parts.

Examples:

- интернет-магазин -> "интернет-" + магазин
- человек-гора -> человек + гора

Analyzer units that analyzes non-word tokes

class pymorphy2.units.by_shape.LatinAnalyzer(morph)
This analyzer marks latin words with "LATN" tag. Example: "pdf" -> LATN

class pymorphy2.units.by_shape.NumberAnalyzer(morph)
This analyzer marks numbers with "NUMB" tag. Example: "12" -> NUMB

Примечание: Don't confuse it with "NUMR": "тридцать" -> NUMR

class pymorphy2.units.by_shape.PunctuationAnalyzer(morph)
This analyzer tags punctuation marks as "PNCT". Example: "," -> PNCT

Tagset

Utils for working with grammatical tags.

class pymorphy2.tagset.OpencorporaTag(tag)

Wrapper class for OpenCorpora.org tags.

> Предупреждение: In order to work properly, the class has to be globally initialized with actual grammemes (using _init_grammemes method).
> Pymorphy2 initializes it when loading a dictionary; it may be not a good idea to use this class directly. If possible, use morph_analyzer.TagClass instead.

Example:

```
>>> from pymorphy2 import MorphAnalyzer
>>> morph = MorphAnalyzer()
>>> Tag = morph.TagClass # get an initialzed Tag class
>>> tag = Tag('VERB,perf,tran plur,impr,excl')
>>> tag
OpencorporaTag('VERB,perf,tran plur,impr,excl')
```

Tag instances have attributes for accessing grammemes:

```
>>> print(tag.POS)
VERB
>>> print(tag.number)
plur
>>> print(tag.case)
None
```

Available attributes are: POS, animacy, aspect, case, gender, involvement, mood, number, person, tense, transitivity and voice.

You may check if a grammeme is in tag or if all grammemes from a given set are in tag:

```
>>> 'perf' in tag
True
>>> 'nomn' in tag
False
>>> 'Geox' in tag
False
>>> set(['VERB', 'perf']) in tag
True
>>> set(['VERB', 'perf', 'sing']) in tag
False
```

In order to fight typos, for unknown grammemes an exception is raised:

```
>>> 'foobar' in tag
Traceback (most recent call last):
...
ValueError: Grammeme is unknown: foobar
>>> set(['NOUN', 'foo', 'bar']) in tag
Traceback (most recent call last):
...
ValueError: Grammemes are unknown: {'bar', 'foo'}
```

This also works for attributes:

```
>>> tag.POS == 'plur'
Traceback (most recent call last):
...
ValueError: 'plur' is not a valid grammeme for this attribute.
```

classmethod fix_rare_cases(grammemes)
: Replace rare cases (loc2/voct/...) with common ones (loct/nomn/...).

grammemes
: A frozenset with grammemes for this tag.

updated_grammemes(required)
: Return a new set of grammemes with required grammemes added and incompatible grammemes removed.

Command-Line Interface

Usage:

```
pymorphy dict compile <DICT_XML> [--out <PATH>] [--force] [--verbose] [--min_ending_freq <NUM>] [--min_paradigm_popula
pymorphy dict download_xml <OUT_FILE> [--verbose]
pymorphy dict mem_usage [--dict <PATH>] [--verbose]
pymorphy dict make_test_suite <XML_FILE> <OUT_FILE> [--limit <NUM>] [--verbose]
pymorphy dict meta [--dict <PATH>]
pymorphy prob download_xml <OUT_FILE> [--verbose]
pymorphy prob estimate_cpd <CORPUS_XML> [--out <PATH>] [--min_word_freq <NUM>]
pymorphy _parse <IN_FILE> <OUT_FILE> [--dict <PATH>] [--verbose]
pymorphy -h | --help
pymorphy --version
```

Options:

```
-v --verbose                        Be more verbose
-f --force                          Overwrite target folder
-o --out <PATH>                     Output folder name [default: dict]
--limit <NUM>                       Min. number of words per gram. tag [default: 100]
--min_ending_freq <NUM>             Prediction: min. number of suffix occurances [default: 2]
--min_paradigm_popularity <NUM>     Prediction: min. number of lexemes for the paradigm [default: 3]
--max_suffix_length <NUM>           Prediction: max. length of prediction suffixes [default: 5]
--min_word_freq <NUM>               P(t|w) estimation: min. word count in source corpus [default: 1]
--dict <PATH>                       Dictionary folder path
```

Utilities for OpenCorpora Dictionaries

class pymorphy2.opencorpora_dict.wrapper.Dictionary(path)
: OpenCorpora dictionary wrapper class.

build_normal_form(para_id, idx, fixed_word)
: Build a normal form.

build_paradigm_info(para_id)
: Return a list of

    (prefix, tag, suffix)

    tuples representing the paradigm.

build_stem(paradigm, idx, fixed_word)
: Return word stem (given a word, paradigm and the word index).

build_tag_info(para_id, idx)
: Return tag as a string.

iter_known_words(prefix=u'')
Return an iterator over (word, tag, normal_form, para_id, idx) tuples with dictionary words that starts with a given prefix (default empty prefix means "all words").

word_is_known(word, strict_ee=False)
Check if a word is in the dictionary. Pass strict_ee=True if word is guaranteed to have correct е/ё letters.

Примечание: Dictionary words are not always correct words; the dictionary also contains incorrect forms which are commonly used. So for spellchecking tasks this method should be used with extra care.

Various Utilities

pymorphy2.tokenizers.simple_word_tokenize(text)
Split text into tokens. Don't split by hyphen.

pymorphy2.shapes.is_latin(token)
Return True if all token letters are latin and there is at least one latin letter in the token:

```
>>> is_latin('foo')
True
>>> is_latin('123-FOO')
True
>>> is_latin('123')
False
>>> is_latin(':)')
False
>>> is_latin('')
False
```

pymorphy2.shapes.is_punctuation(token)
Return True if a word contains only spaces and punctuation marks and there is at least one punctuation mark:

```
>>> is_punctuation(', ')
True
>>> is_punctuation('..!')
True
>>> is_punctuation('x')
False
>>> is_punctuation(' ')
False
>>> is_punctuation('')
False
```

pymorphy2.shapes.is_roman_number(token)
Return True if token looks like a Roman number:

```
>>> is_roman_number('II')
True
>>> is_roman_number('IX')
True
>>> is_roman_number('XIIIII')
False
>>> is_roman_number('')
False
```

pymorphy2.shapes.restore_word_case(word, example)
Make the word be the same case as an example:

```
>>> restore_word_case('bye', 'Hello')
'Bye'
>>> restore_word_case('half-an-hour', 'Minute')
'Half-An-Hour'
>>> restore_word_case('usa', 'IEEE')
'USA'
>>> restore_word_case('pre-world', 'anti-World')
'pre-World'
>>> restore_word_case('123-do', 'anti-IEEE')
'123-DO'
>>> restore_word_case('123--do', 'anti--IEEE')
'123--DO'
```

In the alignment fails, the reminder is lower-cased:

```
>>> restore_word_case('foo-BAR-BAZ', 'Baz-Baz')
'Foo-Bar-baz'
>>> restore_word_case('foo', 'foo-bar')
'foo'
```

pymorphy2.utils.combinations_of_all_lengths(it)
Return an iterable with all possible combinations of items from it:

```
>>> for comb in combinations_of_all_lengths('ABC'):
...     print("".join(comb))
A
B
C
AB
AC
BC
ABC
```

pymorphy2.utils.download_bz2(url, out_fp, chunk_size=262144, on_chunk=<function <lambda> at 0x2b67758>)
Download a bz2-encoded file from url and write it to out_fp file.

pymorphy2.utils.json_read(filename, **json_options)
Read an object from a json file filename

pymorphy2.utils.json_write(filename, obj, **json_options)
Create file filename with obj serialized to JSON

pymorphy2.utils.largest_group(iterable, key)
Find a group of largest elements (according to key).

```
>>> s = [-4, 3, 5, 7, 4, -7]
>>> largest_group(s, abs)
[7, -7]
```

pymorphy2.utils.longest_common_substring(data)
Return a longest common substring of a list of strings:

```
>>> longest_common_substring(["apricot", "rice", "cricket"])
'ric'
>>> longest_common_substring(["apricot", "banana"])
'a'
```

```
>>> longest_common_substring(["foo", "bar", "baz"])
''
```

See http://stackoverflow.com/questions/2892931/.

pymorphy2.utils.word_splits(word, min_reminder=3, max_prefix_length=5)
Return all splits of a word (taking in account min_reminder and max_prefix_length).

1.3.4 Первоначальный формат словарей (отброшенный)

> Предупреждение: Этот формат словарей в pymorphy2 не используется; описание - документация по менее удачной попытке организовать словари.
> Первоначальная реализация доступна в одном из первых коммитов. Рассматривайте описанное ниже как бесполезный на практике "исторический" документ.

В публикации по mystem был описан способ упаковки словарей с использованием 2 trie для "стемов" и "окончаний". В первом прототипе pymorphy2 был реализован схожий способ; впоследствие я заменил его на другой.

Этот первоначальный формат словарей в моей реализации обеспечивал скорость разбора порядка 20-60тыс слов/сек (без предсказателя) при потреблении памяти 30М (с использованием datrie), или порядка 2-5 тыс слов/сек при потреблении памяти 5М (с использованием marisa-trie).

Идея была в том, что слово просматривается с конца, при этом в первом trie ищутся возможные варианты разбора для данных окончаний; затем для всех найденных вариантов окончаний "начала" слов ищутся во втором trie; в результате возвращаются те варианты, где для "начала" и "конца" есть общие способы разбора.

Основной "затык" в производительности был в том, что для каждого слова требовалось искать общие для начала и конца номера парадигм. Это задача о пересечении 2 множеств, для которой мне не удалось найти красивого решения. Питоний set использовать было нельзя, т.к. это требовало очень много памяти.

Лучшее, что получалось - id парадигм хранились в 2 отсортированных массивах, а их пересечение находилось итерацией по более короткому массиву и "сужающимся" двоичным поиском по более длинному (параллельная итерация по обоим массивам на конкретных данных оказывалась всегда медленнее).

В pymorphy2 я в итоге решил использовать другой формат словарей, т.к.

- другой формат проще;
- алгоритмы работы получаются проще;
- скорость разбора получается больше (порядка 100-200 тыс слов/сек без предсказателя) при меньшем потреблении памяти (порядка 15М).

Но при этом первоначальный формат потенциально позволяет тратить еще меньше памяти; некоторые способы ускорения работы с ним еще не были опробованы.

Уменьшение размера массивов, как мне кажется - наиболее перспективный тут способ ускорения. Для уменьшения размеров сравниваемых массивов требуется уменьшить количество парадигм (например, "вырожденных" с пустым стемом).

Выделение парадигм

Изначально в словаре из OpenCorpora нет понятия "парадигмы" слова. Парадигма - это таблица форм какого-либо слова, образец для склонения или спряжения.

В pymorphy2 выделенные явным образом парадигмы слов необходимы для того, чтоб склонять неизвестные слова - т.к. при этом нужны образцы для склонения.

Пример исходной леммы:

```
375080
ЧЕЛОВЕКОЛЮБИВ   100
ЧЕЛОВЕКОЛЮБИВА  102
ЧЕЛОВЕКОЛЮБИВО  105
ЧЕЛОВЕКОЛЮБИВЫ  110
```

Парадигма (пусть будет номер 12345):

```
""      100
"А"     102
"О"     105
"Ы"     110
```

Вся лемма при этом "сворачивается" в "стем" и номер парадигмы:

```
"ЧЕЛОВЕКОЛЮБИ" 12345
```

Примечание: Для одного "стема" может быть несколько допустимых парадигм.

Прилагательные на ПО-

В словарях у большинства сравнительных прилагательных есть формы на ПО-:

```
375081
ЧЕЛОВЕКОЛЮБИВЕЕ COMP,Qual V-ej
ПОЧЕЛОВЕКОЛЮБИВЕЕ       COMP,Qual Cmp2
ПОЧЕЛОВЕКОЛЮБИВЕЙ       COMP,Qual Cmp2,V-ej
```

Можно заметить, что в этом случае форма слова определяется не только тем, как слово заканчивается, но и тем, как слово начинается. Алгоритм с разбиением на "стем" и "окончание" приведет к тому, что все слово целиком будет считаться окончанием, а => каждое сравнительное прилагательное породит еще одну парадигму. Это увеличивает общее количество парадигм в несколько раз и делает невозможным склонение несловарных сравнительных прилагательных, поэтому в pymorphy2 парадигма определяется как "окончание", "номер грам. информации" и "префикс".

Пример парадигмы для "ЧЕЛОВЕКОЛЮБИВ":

```
""      100     ""
"А"     102     ""
"О"     105     ""
"Ы"     110     ""
```

Пример парадигмы для "ЧЕЛОВЕКОЛЮБИВЕЕ":

```
""      555     ""
""      556     "ПО"
""      557     "ПО"
```

Примечание: Сейчас обрабатывается единственный префикс - "ПО". В словарях, похоже, нет других префиксов, присущих только отдельным формам слова в пределах одной леммы.

Упаковка “стемов”

“Стемы” - строки, основы лемм. Для их хранения используется структура данных trie (с использованием библиотеки datrie), что позволяет снизить потребление оперативной памяти (т.к. некоторые общие части слов не дублируются) и повысить скорость работы (т.к. в trie можно некоторые операции - например, поиск всех префиксов данной строки - можно выполнять значительно быстрее, чем в хэш-таблице).

Ключами в trie являются стемы (перевернутые), значениями - список с номерами допустимых парадигм.

Упаковка tuple/list/set

Для каждого стема требуется хранить множество id парадигм; обычно это множества из небольшого числа int-элементов. В питоне накладные расходы на set() довольно велики:

```
>>> import sys
>>> sys.getsizeof({})
280
```

Если для каждого стема создать даже по одному пустому экземпляру set, это уже займет порядка 80M памяти. Поэтому set() не используется; сначала я заменил их на tuple с отсортированными элементами. В таких tuple можно искать пересечения за O(N+M) через однопроходный алгоритм, аналогичный сортировке слиянием, или за O(N*log(M)) через двоичный поиск.

Но накладные расходы на создание сотен тысяч tuple с числами тоже велики, поэтому в pymorphy 2 они упаковываются в одномерный массив чисел (array.array).

Пусть у нас есть такая структура:

```
(
    (10, 20, 30),       # 0й элемент
    (20, 40),           # 1й элемент
)
```

Она упакуется в такой массив:

```
array.array([3, 10, 20, 30, 2, 20, 40])
```

Сначала указывается длина данных, затем идет сами данные, потом опять длина и опять данные, и т.д. Для доступа везде вместо старых индексов (0й элемент, 1й элемент) используются новые: 0й элемент, 4й элемент. Чтоб получить исходные цифры, нужно залезть в массив по новому индексу, получить длину N, и взять следующие N элементов.

Итоговый формат данных

Таблица с грам. информацией

```
['tag1', 'tag2', ...]
```

tag<N> - набор грам. тегов, например NOUN,anim,masc sing,nomn.

Этот массив занимает где-то 0.5M памяти.

Парадигмы

```
[
    (
        (suffix1, tag_index1, prefix1),
        (suffix2, tag_index2, prefix2),
        ...
    ),
    (
        ...
]
```

suffix<N> и prefix<N> - это строки с окончанием и префиксом (например, "ЫЙ" и); tag_index<N> - индекс в таблице с грам. информацией.

Парадигмы занимают примерно 7-8M памяти.

Примечание: tuple в парадигмах сейчас не упакованы в линейные структуры; упаковка должна уменьшить потребление памяти примерно на 3M.

Стемы

Стемы хранятся в 2 структурах:

- array.array с упакованными множествами номеров возможных парадигм для данного стема:

  ```
  [length0, para_id0, para_id1, ..., length1, para_id0, para_id1, ...]
  ```

- и trie с ключами-строками и значениями-индексами в массиве значений:

  ```
  datrie.BaseTrie(
      'stem1': index1,
      'stem2': index2,
      ...
  )
  ```

"Окончания"

Для каждого "окончания" хранится, в каких парадигмах на каких позициях оно встречается. Эта информация требуется для быстрого поиска нужного слова "с конца". Для этого используются 3 структуры:

- array.array с упакованными множествами номеров возможных парадигм для данного окончания:

  ```
  [length0, para_id0, para_id1, ..., length1, para_id0, para_id1, ...]
  ```

  В отличие от аналогичного множества для стемов, номера парадигм могут повторяться в пределах окончания.

- array.array с упакованными множествами индексов в пределах парадигмы:

  ```
  [length0, index0, index1, ..., length1, index0, index1, ...]
  ```

Этот массив работает “вместе” с предыдущим, каждому элементу отсюда соответствует элемент оттуда - совместно они предоставляют информацию о возможных номерах форм в парадигме для всех окончаний.

- trie с ключами-строками и значениями-индексами:

```
datrie.BaseTrie(
    'suff1': index1,
    'suff2': index2,
    ...
)
```

По индексу index<N> можно из предудыщих 2х массивов получить наборы форм для данного окончания.

Примечание: Длины хранятся 2 раза. Может, это можно как-то улучшить?

1.4 Терминология

лексема Набор всех форм одного слова. Например, “ёж”, “ежи” и “ежам” входят в одну лексему. [1]

лемма, нормальная форма слова Каноническая форма слова (например, форма единственного числа, именительного падежа для существительных). [2]

граммема Значение какой-либо грамматической характеристики слова. Например, “множественное число” или “деепричастие”. Множество всех граммем, характеризующих данное слово, образует тег.

См. также: Обозначения для граммем.

тег Набор граммем, характеризующих данное слово. Например, для слова “ежам” тегом может быть 'NOUN,anim,masc plur,datv'.

парадигма, словоизменительная парадигма Образец для склонения или спряжения; правила, согласно которым можно получить все формы слов в лексеме для данного стема.

В pymorphy2 для каждого слова в словаре указано, по каким парадигмам это слово могло быть образовано; pymorphy2 также умеет предсказывать парадигму для слов, отсутствующих в словаре.

стем Неизменяемая часть слова.

- genindex
- modindex
- search

Исходный код - на github или bitbucket. Если заметили ошибку, то пишите в баг-трекер. Для обсуждения есть гугл-группа; если есть какие-то вопросы - пишите туда.

---

[1] Часто не делается различия между леммой и лексемой, или термин “лемма” употребляется в значении “набор форм слова”. Но, похоже, данное выше определение лексемы все же более стандартное (см., например, см. википедию или Foundations of Statistical Natural Language Processing), поэтому в pymorphy2 набор всех форм слова называется именно лексемой.

[2] В pymorphy1 и в XML-словаре из OpenCorpora слово “лемма” употребляется в значении “лексема”. Чтобы не усугублять путаницу, в pymorphy2 вместо термина “лемма” употребляется термин “нормальная форма слова”, а термин “лемма” не используется совсем.

Глава 2

Цели и задачи

- Поддержка всех возможностей pymorphy (не готово);
- более актуальные и точные словари из OpenCorpora;
- большая скорость работы (50x-500x) при таком же или меньшем потреблении памяти;
- преобразование слов из одной формы в другую между разными частями речи;
- выделение поддержки django в отдельный пакет (не готово);
- полная поддержка буквы ё;
- возможность обновления словарей;
- ранжирование результатов разбора (готово только частично);
- снятие неоднозначности разбора (?) (не готово).

Python Module Index

p

pymorphy2.analyzer, ??
pymorphy2.cli, ??
pymorphy2.opencorpora_dict.wrapper, ??
pymorphy2.shapes, ??
pymorphy2.tagset, ??
pymorphy2.tokenizers, ??
pymorphy2.units.by_analogy, ??
pymorphy2.units.by_hyphen, ??
pymorphy2.units.by_lookup, ??
pymorphy2.units.by_shape, ??
pymorphy2.utils, ??