PARSER GENERATOR

CSC, 2017

Оглавление

Глава 1. Интерфейсы

Tree

Интерфейс деревьев, для парсинга которых генерируется парсер. Задаёт три абстрактных метода и два поля:

```
tag
```

Поле задаёт название или тип корня дерева.

```
classes
```

Поле задаёт список каких-либо атрибутов корня дерева. Необходим для работы $BlackListSelector$, но не для $SimpleSelector$.

```
def get_value(self, tree_path: TreePath) -> str
```

Метод возвращает значение элемента дерева, на который указывает $tree\_path$.

```
def get_elements(self, tree_path: TreePath) -> List[Tree]
```

Метод возвращает список поддеревьев, которые удовлетворяют пути $tree\_path$.

```
def get_iter(self)
```

Переопределение базового метода. Возвращает итератор по всем элементам дерева.

```
def cssselect(self, attribute: str) -> List(Tree)
```

Опционально. Метод возвращает список всех вершин дерева, удовлетворяющих CSS селектору $attribute$. Необходим для работы $BlackListSelector$.

TreePath

Интерфейс путей в дереве. Задаёт основные операции, которые можно производить с путями:

```
def get_relative_path(self, tree_path) -> TreePath
```

Метод возвращает путь относительно пути $tree\_path$. Необходима поддержка лишь случая, когда $tree\_path$ является префиксом текущего пути. Без обобщения на пути вида «../../».

```
def get_common_prefix(self, tree_path, in_block=False) ->
  TreePath
```

Метод возвращает наибольший общий префикс путей. Если флак in_block выставлен, то для путей производится их обощение по индексам. Иначе определяется точное совпадение.

```
def len(self) -> int
```

Метод возвращает длину пути – целое число.

```
def drop_for_len(self, len) -> TreePath
```

Обрезание пути до заданной длины.

```
def concat(self, tree_path) -> TreePath
```

Конкатенация путей. Входной параметр должен содержать относительный путь.

Markup

Интерфейс разметки. Необходимо определить три поля и один метод:

file

Относительный путь до файла, которому соответствует разметка.

type

Тип разметки, поддерживаемый классом $MarkupTypeRegistry$.

components

Список элементов разметки, определённых в заданном файле.

def add(self, component)

Метод добавления новой компоненты в список $components$.

Algorithm

Интерфейс основного алгоритма. Задаёт два метода, которые определяют схему взаимодействия с ним:

def learn(self, markup_list: List[Markup])

Метод обучения парсера на списке разметок. На вход принимает список разметок одного типа.

def parse(self, raw_page: **str**) -> ParserResult:

Метод для парсинга страницы. Следует вызывать после вызова обучения парсера. Возвращает элемент типа $ParserResult$.

Selector

Интерфейс стратегии разрешения коллизий разных типов компонент, расположенных по одному пути.

```
def learn(self, algorithm: Algorithm, markup_list: List[
    Markup]) -> None
```

Метод обучения селектора на данных алгоритма и списке разметок.

```
def get_iter(self, **kwargs)
```

Переопределение базового метода. Возвращает итератор по компонентам в порядке разрешения коллизий.

Глава 2. Реализации

MarkupTypeRegistry

Singleton определяющий поддерживаемые типы деревьев. На данный момент имеется поддержка типов «HTMLTree» и «JSONTree», задающих деревья на HTML и JSON структурах соответственно. Содержит единственный метод:

```
def get_tree(self, markup_type, raw_page)
```

Метод возвращает дерево типа с названием $markup\_type$, построенного по строке raw_page.

HTMLTree

Дерево структуры HTML, представляет из себя обёртку над деревом $lxml.html$ с добавлением реализации методов интерфейса $Tree$.

```
def __init__(self, **kwargs)
```

Может принимать raw_page или $html\_tree$. В первом случае с помощью методов библиотеки $lxml.html$ из HTML строки строится дерево, во втором – сохраняется указатель на переданное дерево типа $lxml.html$. Поля tag и $classes$ также берутся из соответствующих полей типа $lxml.html$.

```
def cssselect(self, attribute)
```

Используется одноимённый метод библиотеки $lxml.html$.

```
def get_iter(self)
```

Возвращает экземпляр класса $TagIterator$, представляющего собой обёртку над методом $iter$ класса $lxml.html$.

```
def get_elements(self, html_path)
```

Используется метод $xpath$ класса $lxml.html$, все элементы результата которого оборачиваются в $HTMLTree$.

```
def get_value(self, html_path)
```

Берётся первый элемент дерева, удовлетворяющий пути $html\_path$. Для атрибутов «href», «title», «src» возвращается значение соответствующего атрибута тега. Для атрибута «style» возвращается значение после «//». Во всех остальных случаях возвращается содержимое тега с помощью метода $text\_content$.

HTMLPath

Пути в $HTMLTree$, представляющие из себя XPath до HTML тега в дереве и название атрибута тега.

```
def split_xpath(self)
```

Разделяет путь на компоненты по одной вершине. Для каждой компоненты хранится кортеж-пара (тег, индекс тега). Нумерация индексов с 1. Если индекс отсутствует, то значение равно 0. Возвращает список кортежей.

```
def merge_xpath(self, extract_list, relative=False)
```

Склеивает относительный или абсолюный путь из списка кортежей формата, возвращаемого $split\_xpath$.

```
def get_common_prefix(self, tree_path, in_block=False)
```

Наибольший общий префикс находим следующим образом: оба пути разделяем на компоненты и двумя указателями идём пока теги совпадают. Если in_block выставлен в False, то разница индексов не учитывается, а у итогового кортежа выставляется значение 0. Для путей одного блока сравнение идёт с учётом индексов. Итоговый кортеж склеивается в путь. В качестве атрибута берётся атрибут параметра $tree\_path$.

```
def get_relative_path(self, tree_path)
```

Подразумевается, что $tree\_path$ является префиксом пути, тогда оба пути разделяются и берётся только хвост текущего пути, который склеивается в итоговый путь с флагом $relative = True$.

```
def len(self)
```

Возвращает длину списка компонент пути.

```
def drop_for_len(self, len)
```

Склеиваются лишь первые len элементов списка компонент пути. Поле $attr$ выставляется в None.

```
def concat(self, tree_path)
```

Объединение строк путей с проверкой на наличие «.» в начале для относительного пути. Атрибут берётся из параметра $tree\_path$.

JSONTree

Дерево структуры JSON, представляет из себя обёртку над объектами Python, полученными из JSON представления. Реализация идентична реализации HTMLTree с той разницей, что вместо XPath используется JSONPath одноимённой библиотеки. А также поле $classes$ и метод $cssselect$ возвращают пустой список.

JSONPath

Пути в $JSONTree$, представляющие из себя JSONPath. Реализация аналогична реализации $HTMLPath$.

Component

Класс определяющий базовые поля компонент результата парсинга и реализующий серилизацию и сравнение объектов. К базовым полям относятся: $type$, $alignment$, $page\_url$, $title$. Серилизация реализована с помощью $jsonpickle$. А сравнение – сравнением словарей, так как объекты «плоские».

ParserResult

Класс результатов парсинга, представляющих из себя список объектов типа $Component$. Реализована серилизация с помощью $jsonpickle$.

```
def add(self, component)
```

Добавляет компоненту в список $components$.

```
def count(self, component_type=None)
```

Возвращает размер списка. Если задан $component\_type$, то возвращает количество компонент данного типа.

SearchMarkup

Тривиальная имплементация интерфейса $Markup$ с серилизацией через $jsonpickle$.

SearchMarkupComponent

Расширение класса $Component$ до базового класса всех компонент поисковой разметки.

SearchMarkupSearchResult, SearchMarkupAdv,

SearchMarkupWizardImage, SearchMarkupWizardNews

Специализации компонент поисковой выдачи: документ, реклама, колдунщики картинок и новостей, которые задают поля типа компоненты и определяют специфические поля.

Algorithm_v2

Реализация основного алгоритма на базе сжатого бора путей и с поддержкой различных способов разрешения коллизий.

`types`

Список типов компонент, которые встречаются в разметке.

`markup_type`

Тип разметки, согласованный с $MarkupTypeRegistry$.

`root`

Корень бора. Каждая вершина бора – объект типа $Node$ – содержит поля:

- $indexes$ – список индексов типов компонент, которые встречаются по пути, соответствующему данной вершине бора, в соответствии со списком $types$,
- $samples$ – список компонент разметки по одной на каждый тип, согласно списку $indexes$,

- $treepaths$ – список рёбер, выходящих из вершины. Элемент списка – кортеж (объект $TreePath$, объект $Node$) – путь до вершины и сама вершина.

```
def __init__(self, directory, selector)
```

Конструктор принимает директорию с исходниками документов для обучения и объект для разрешения коллизий.

```
def learn(self, markup_list)
```

Метод построения парсера.

Предполагается получение списка разметок одного типа, поэтому тип разметки определяется по первому элементу списка.

Далее для каждой компоненты всех разметок входного списка выполняется следующая процедура: для всех полей компоненты (если поле является списком, то для каждого элемента списка) строится единый общий префикс с флагом in_block. Таким образом находится путь до объединяющего узла компоненты. Параллельно с этим определяется тип компоненты и формируется список $types$. Дальше эта компонента добавляется в бор по полученному пути методом $add\_component$.

После построение бора и определения всех остальных внутренних полей вызывается обучение $selector'$а.

```
def add_component(self, node, treepath, index_of_type,
    sample)
```

Рекурсивная функция добавления компоненты в бор.

Принимает текущую вершину бора; суффикс пути, который ещё остался; индекс типа компоненты, согласно списку $types$; добавляемую компоненту.

Если путь пустой, то делается проверка на наличие компоненты такого типа в текущую вершину и, при необходимости, добавляется в списки $samples$ и $indexes$. И выполнение функции заканчивается.

Среди всех путей, исходящих из текущей вершины, ищется такой, чтобы общий префикс с пришедшим в параметрах был не пуст. Причём общий префикс ищется без учёта индексов узлов в дереве, тем самым в боре строятся обобщающие пути, единые для разных компонент. Если найденный путь короче пришедшего суффикса, то рекурсивно переходим в вершину бора, в которую ведёт этот путь и продолжаем процесс. Иначе делим ребро, добавляя новую вершину в бор и размещая в этой вершине нашу компоненту.

Если же ни один путь не подошёл, то в бор добавляется новая вершина с путём до неё равным текущему суффиксу.

При передачи на следующие уровни рекурсии компонента проходит через функцию $get\_relative\_component$, которая для всех полей компоненты делает пути относительными к тому, который уже получен в боре.

```
def parse(self, raw_page)
```

Головной метод парсинга строки, которые создает объект $ParserResult$ для результата работы, строит дерево из пришедшей строки и вызывает обход бора в глубину – dfs.

```
def dfs(self, node, tree, parser_result, check=False,
   current_path=None)
```

Рекурсивный обход бора в глубину с параллельным движение по дереву.

Принимает текущую вершину бора; текущую вершину дерева; объект результата; флаг проверки; текущий путь до вершины дерева. Если выставлен флаг $check$, то метод работает в режиме проверки и пытается найти компоненту $parser\_result$ и её путь в дереве.

Первым делом пытаемся спарсить какую-либо компоненту, имеющую такой обобщённый путь, итерируясь по них в порядке, заданном $selector'$ом. Если получилось удачно, то, в зависимости от флага $check$ либо добавляем компоненту к результату, либо после проверки на равенство сообщаем успешный результат поиска и возвращаем вершину дерева, путь и индекс типа компоненты.

А дальше перебираем все обобщённые пути бора и реальные пути дерева, соответствующие путям бора, и рекурсивно переходим в следующие вершины.

```
def parse_component(self, element, sample)
```

Данный метод пытается, начиная с вершины дерева $element$, спарсить компоненту $sample$, перебирая все поля. Поддерживаются поля трёх типов: строка (просто копируется), объект $TreePath$ (выполняется get_value для него), список объектов $TreePath$ (из них строится обобщающий путь, а потом парсятся все элементы, которые удовлетворяют этому пути).

```
def get_substitution(self, tree, markup)
```

Метод, выполняющий подстановку значений вместо путей в разметке. Для всех полей каждой компоненты разметки выполняется get_value (если тип поля строка, то она просто копируется).

Метод требуется для $BlackListSelector$.

```
def get_element_for_parser_component(self, parser_component
    , tree)
```

Вспомогательный метод для $BlackListSelector$, который запускает dfs с правильными параметрами для нахождения заданной компоненты в дереве.

SimpleSelector

Тривиальная имплементация интерфейса $Selector$, которая разрешает коллизии в порядке уменьшения количества полей у компонент.

```
def get_iter(self, **kwargs)
```

Метод ожидает словарь с элементом «node» – вершиной бора. Строится список из индексов на индексы типов компонент, соответствующих по-

лученной вершине бора. Затем этот список сортируется в порядке уменьшения размера словаря компонент. Получившийся список возвращается в качестве итератора.

BlackListSelector

Имплементация интерфейса $Selector$ с построение чёрных списков элементов поддерева и их CSS селекторов для типов компонент, позволяющих различить компоненты разного типа.

```
def get_iter(self, **kwargs)
```

Метод ожидает словарь с элементом «node» – вершиной бора и «tree» – деревом входной строки. Как и в $SimpleSelector$ строится список из индексов на индексы типов компонент, соответствующих полученной вершине бора и не имеющих в чёрных списках элементов дерева $tree$. Затем этот список сортируется в порядке уменьшения размера словаря компонент. Получившийся список возвращается в качестве итератора.

```
def is_not_black(self, element, element_type)
```

Для каждого CSS селектора из чёрного списка компоненты типа $element\_type$ ищутся соответствующие им вершины в дереве $element$. Если находится хотя бы один, то эта компонента из чёрного списка и не должна парситься в этом дереве. Иначе возвращается $True$.

```
def learn(self, algorithm, markup_list)
```

Для каждой разметки из входного списка строится два результата – обученным алгоритмом и подстановкой значений из дерева в разметку. Если результаты различаются, то находятся лишние (либо неверно определённый тип) компоненты, которые были распознаны алгоритмом, и их пути в дереве. Для каждой такой компоненты вызывается метод $add\_black\_for\_element$.

```
def add_black_for_element(self, algorithm, element,
   element_path, index_of_element_type, markup_list)
```

В первую очередь составляется список кортежей (тег, класс) всех элементов поддерева.

Затем для каждой разметки из входного списка находятся компоненты того типа, который хотим отличить. Обрезанием полного пути до заголовка компоненты, находим путь до объединяющего элемента компоненты. По этому в дереве получаем вершину и проверяем, какие кортежи (тег, класс) имеются в поддереве этой вершины. Для всех найденных в списке $list\_flag$ отмечаем, что эта пара не походит.

В заключении, находим первую кортеж-пару, которая не была исключена и ещё не добавлена в чёрный список искомого типа компоненты, и добавляем её в $blacks$.