Domain Specific Languages in Python

Siddharta Govindaraj
siddharta@silverstripessoftware.com
What are DSLs?

Specialized mini-languages for specific problem domains that make it easier to work in that domain
Example: SQL

SQL is a mini language specialized to retrieve data from a relational database
Example: Regular Expressions

Regular Expressions are mini languages specialized to express string patterns to match
def is_ip_address(ip_address):
    components = ip_address_string.split(".")
    if len(components) != 4: return False
    try:
        int_components = [int(component) for component in components]
    except ValueError:
        return False
    for component in int_components:
        if component < 0 or component > 255:
            return False
    return True
Life With Regular Expressions

def is_ip(ip_address_string):
    match = re.match(r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$", ip_address_string)
    if not match: return False
    for component in match.groups():
        if int(component) < 0 or int(component) > 255:
            return False
    return True
The DSL that simplifies our life

^\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}$
Why DSL - Answered

When working in a particular domain, write your code in a syntax that fits the domain.

When working with patterns, use RegEx

When working with RDBMS, use SQL

When working in your domain – create your own DSL
The two types of DSLs

External DSL – The code is written in an external file or as a string, which is read and parsed by the application
The two types of DSLs

**Internal DSL** – Use features of the language (like metaclasses) to enable people to write code in python that resembles the domain syntax.
Creating Forms – No DSL

<form>
<label>Name:</label><input type="text" name="name"/>
<label>Email:</label><input type="text" name="email"/>
<label>Password:</label><input type="password" name="name"/>
</form>
Creating Forms – No DSL

– Requires HTML knowledge to maintain
– Therefore it is not possible for the end user to change the structure of the form by themselves
Creating Forms – External DSL

UserForm
name->CharField label: Username
email->EmailField label: Email Address
password->PasswordField

This text file is parsed and rendered by the app
Creating Forms – External DSL

+ Easy to understand form structure
+ Can be easily edited by end users
– Requires you to read and parse the file
Creating Forms – Internal DSL

class UserForm(forms.Form):
    username = forms.RegexField(regex=r'^\w+$',
                                max_length=30)
    email = forms.EmailField(max_length=75)
    password =
        forms.CharField(widget=forms.PasswordInput())

Django uses metaclass magic to convert this syntax to an easily manipulated python class
Creating Forms – Internal DSL

+ Easy to understand form structure
+ Easy to work with the form as it is regular python
+ No need to read and parse the file
– Cannot be used by non-programmers
– Can sometimes be complicated to implement
– Behind the scenes magic → debugging hell
Creating an External DSL

UserForm
name: CharField -> label: Username size: 25
email: EmailField -> size: 32
password: PasswordField

Lets write code to parse and render this form
Options for Parsing

Using string functions → You have to be crazy

Using regular expressions →

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. - Jamie Zawinski

Writing a parser → ✓ (we will use PyParsing)
Step 1: Get PyParsing

```
pip install pyparsing
```
Step 2: Design the Grammar

form ::= form_name newline field+
field ::= field_name colon field_type [arrow property+] 
property ::= key colon value
form_name ::= word
field_name ::= word
field_type ::= CharField | EmailField | PasswordField
key ::= word
value ::= alphanumeric+
word ::= alpha+
newline ::= \n
colon ::= :
arrow ::= ->
Quick Note

Backus-Naur Form (BNF) is a syntax for specifying grammars
Step 3: Implement the Grammar

```
newline = "\n"
colon = "::
arrow = "-->"
word = Word(alphas)
key = word
value = Word(alphanums)
field_type = oneOf("CharField EmailField PasswordField")
field_name = word
form_name = word
field_property = key + colon + value
field = field_name + colon + field_type +
        Optional(arrow + OneOrMore(field_property)) + newline
form = form_name + newline + OneOrMore(field)
```
Quick Note

PyParsing itself implements a neat little internal DSL for you to describe the parser grammar.

Notice how the PyParsing code almost perfectly reflects the BNF grammar.
PyParsing has neatly parsed our form input into tokens. That's nice, but we can do more.
Step 4: Suppressing Noise Tokens

newline = Suppress("\n")
colon = Suppress(":")
arrow = Suppress("->")
Output

```
> print form.parseString(input_form)

['UserForm', 'name', 'CharField', 'label', 'Username', 'size', '25', 'email', 'EmailField', 'size', '25', 'password', 'PasswordField']
```

All the noise tokens are now removed from the parsed output
Step 5: Grouping Tokens

field_property = Group(key + colon + value)

field = Group(field_name + colon + field_type +
Group(Optional(arrow + OneOrMore(field_property))) +
newline)
Output

> print form.parseString(input_form)

['UserForm',
 ['name', 'CharField',
  [['label', 'Username'], ['size', '25']]],
 ['email', 'EmailField',
  [['size', '25']]],
 ['password', 'PasswordField',[]]]

Related tokens are now grouped together in a list
Step 6: Give Names to Tokens

form_name = word.setResultsName("form_name")
field = Group(field_name + colon + field_type +
    Group(Optional(arrow + OneOrMore(field_property))) +
    newline).setResultsName("form_field")
Output

> parsed_form = form.parseString(input_form)
> print parsed_form.form_name

UserForm

> print parsed_form.fields[1].field_type

EmailField

Now we can refer to parsed tokens by name
Step 7: Convert Properties to Dict

def convert_prop_to_dict(tokens):
    prop_dict = {}
    for token in tokens:
        prop_dict[token.property_key] =
            token.property_value
    return prop_dict

field = Group(field_name + colon + field_type +
    Optional(arrow + OneOrMore(field_property))
    .setParseAction(convert_prop_to_dict) +
    newline).setResultsName("form_field")
Output

> print form.parseString(input_form)

['UserForm',
 ['name', 'CharField',
  {'size': '25', 'label': 'Username'}],
 ['email', 'EmailField',
  {'size': '32'}],
 ['password', 'PasswordField', {}]]

Sweet! The field properties are parsed into a dict
Step 7: Generate HTML Output

We need to walk through the parsed form and generate an HTML string out of it.
def get_field_html(field):
    properties = field[2]
    label = properties["label"] if "label" in properties else field.field_name
    label_html = "<label>" + label + "</label>"
    attributes = {"name":field.field_name}
    attributes.update(properties)
    if field.field_type == "CharField" or field.field_type == "EmailField":
        attributes["type"] = "text"
    else:
        attributes["type"] = "password"
    if "label" in attributes:
        del attributes["label"]
    attributes_html = " ".join([name+'="'+value+'"' for name, value in attributes.items()])
    field_html = "<input " + attributes_html + "/>"
    return label_html + field_html + "<br/>"

def render(form):
    fields_html = " ".join([get_field_html(field) for field in form.fields])
    return "<form id=" + form.form_name.lower() +"">" + fields_html + "</form>"
Output

```javascript
> print render(form.parseString(input_form))

<form id='userform'>
  <label>Username</label>
  <input type='text' name='name' size='25'/><br/>
  <label>email</label>
  <input type='text' name='email' size='32'/><br/>
  <label>password</label>
  <input type='password' name='password'/><br/>
</form>
```
It works, but....

Yuck!

The output rendering code is an UGLY MESS
Wish we could do this...

```python
> print Form(CharField(name="user", size="25", label="ID"),
        id="myform")

<form id='myform'>
<label>ID</label>
<input type='text' name='name' size='25'/><br/>
</form>
```

Neat, clean syntax that matches the output domain well. But how do we create this kind of syntax?
Let's create an Internal DSL
class HtmlElement(object):
    default_attributes = {}  
tag = "unknown_tag"

    def __init__(self, *args, **kwargs):
        self.attributes = kwargs
        self.attributes.update(self.default_attributes)
        self.children = args

    def __str__(self):
        attribute_html = " ".join(["{}='{}'".format(name, value) for name, value in 
        self.attributes.items()])

        if not self.children:
            return "<{} {}/>".format(self.tag, attribute_html)
        else:
            children_html = " ".join([str(child) for child in self.children])
            return "<{} {}>{}</{}>".format(self.tag, attribute_html, children_html,
self.tag)
> print HtmlElement(id="test")

<unknown_tag id='test'/>

> print HtmlElement(HtmlElement(name="test"), id="id")

<unknown_tag id='id'><unknown_tag name='test'/></unknown_tag>
class Input(HtmlElement):
    tag = "input"

    def __init__(self, *args, **kwargs):
        HtmlElement.__init__(self, *args, **kwargs)
        self.label = self.attributes["label"] if "label" in self.attributes else self.attributes["name"]

        if "label" in self.attributes:
            del self.attributes["label"]

    def __str__(self):
        label_html = "<label>{}</label>".format(self.label)
        return label_html + HtmlElement.__str__(self) + "<br/>"
> print InputElement(name="username")

<label>username</label><input name='username'/><br/>

> print InputElement(name="username", label="User ID")

<label>User ID</label><input name='username'/><br/>
class Form(HtmlElement):
    tag = "form"

class CharField(Input):
    default_attributes = {"type":"text"}

class EmailField(CharField):
    pass

class PasswordField(Input):
    default_attributes = {"type":"password"}
Now...

```python
> print Form(CharField(name="user",size="25",label="ID"),
    id="myform")

<form id='myform'>
<label>ID</label>
<input type='text' name='name' size='25'/><br/>
</form>
```

Nice!
Step 7 Revisited: Output HTML

def render(form):
    field_dict = {
        "CharField": CharField,
        "EmailField": EmailField,
        "PasswordField": PasswordField
    }
    fields = [field_dict[field.field_type]
              (name=field.field_name, **field[2])
              for field in form.fields]

    return Form(*fields, id=form.form_name.lower())

Now our output code uses our Internal DSL!
INPUT
UserForm
name:CharField -> label:Username size:25
email:EmailField -> size:32
password:PasswordField

OUTPUT
<form id='userform'>
<label>Username</label>
<input type='text' name='name' size='25'/><br/>
<label>email</label>
<input type='text' name='email' size='32'/><br/>
<label>password</label>
<input type='password' name='password'/><br/>
</form>
Get the whole code

Summary

+ DSLs make your code easier to read
+ DSLs make your code easier to write
+ DSLs make it easy to for non-programmers to maintain code
+ PyParsing makes it easy to write External DSLs
+ Python makes it easy to write Internal DSLs