Enaml

A DSL for Declarative UIs

Puneeth Chaganti

Enthought

29 Sep 2012

What is Enaml?

- a library for building beautiful user interfaces with minimal effort
- a declarative language based on Python, heavily inspired by Qt's QML
- the Enaml language is a superset of Python
- uses a constraints based layout system to allowing users to easily express complex UI layouts
- can run on multiple backends (Qt and Wx) and on multiple operating systems

Goals

The project was developed with the following goals in mind

- Integrate well with Traits and Chaco
- Help separate the presentation and content (i.e., MVC)
- Allow a single script to work across multiple widget toolkits when using the default interfaces.
- Be extensible and allow adaptation and addition of the base widgets with little effort.
- Well documented code base that is easy to understand

Example applications

- Image Processing – ˜200 LOC
- Preview App – ˜150 LOC

Simple example

```
from traits.api import HasTraits, Str
import enaml

class Person(HasTraits):
    first_name = Str
    last_name = Str


if __name__ == '__main__':
    john = Person(first_name='John',
                  last_name='Doe')

    with enaml.imports():
        from view import View
        view.show()
```

```
enamldef View(MainWindow):
    attr person
    title = 'Person View'
    Form:
        Label:
            text = 'First Name'
        Field:
            value := person.first_name
    Form:
        Label:
            text = 'Last Name'
        Field:
            value := person.last_name
```

Language Structure

- Enaml is a strict superset of Python
- Any valid Python (2.x) file is a valid Enaml file
- Enaml extends Python with the keyword `enamldef`
- The `enamldef` keyword begins a block of Enaml code which extends Python's standard grammar and scoping rules.

Language Structure

- Enaml components are widget trees with dynamic bindable attributes
- The root of a component derives from another root or a built-in Enaml component and defines a new usable component type

```
enamldef CustomField(Field):
    pass

enamldef ReallyCustomField(Field):
    pass
```

Language Structure

- Tree branches are instances of tree roots or built-in components
- Tree leaves are identical to branches but have no children

```
enamldef MyContainer(Container):
    CustomField:
        pass
    ReallyCustomField:
        pass
    Container:
        Field:
            pass
        PushButton:
            pass
```

Language Structure

- Roots and branches are customized by binding to their attributes

```
enamldef Main(Window):
    title = 'Window Title'
    Field:
        value = 'Field Value'
```

- Roots can be further customized by declaring new attributes and events

```
enamldef Main(Window):
    attr model
    event custom_event
    title = 'Window Title'
    Field:
        value = model.value
```

Language Structure

- The grammar of declaring and `attr` or an `event` supports four different forms
    - `(event|attr) <name>`
    - `(event|attr) <name>: <type>`
    - `(event|attr) <name> <binding>`
    - `(event|attr) <name>: <type> <binding>`

Attribute Binding

- Enaml provides five different operators which can be used to bind Python expressions to component attributes
- The operators provide very powerful introspection and dependency tracking
- Each binding operator has its own behavioral semantics as well as restrictions on what form the Python expressions may take

Attribute Binding - Default

- =
- Left associative
- Single eval, no introspection
- RHS can be any expression

```
enamldef Main(Window):
    attr message = "Hello, world!"
    Container:
        Label:
            text = message
```

Attribute Binding - Subscription

- $<<$
- Left associative
- Evals and assigns on change
- RHS can be any expression

Attribute Binding - Subscription

```
import math

enamldef Main(MainWindow):
    title = 'Slider Example'
    Form:
        Label:
            text = 'Log Value'
        Field:
            value << math.log(val_slider.value)
            read_only = True
        Slider:
            id: val_slider
            tick_interval = 50
            maximum = 1000
            minimum = 1
```

Attribute Binding - Update

- $>>$
- Right associative
- Pushes value on change
- RHS must be an assignable expression

Attribute Binding - Update

```
from traits.api import HasTraits, Str, on_trait_change


class Person(HasTraits):
    name = Str

    @on_trait_change('name')
    def print_name(self):
        print 'name changed', self.name

enamldef Main(Window):
    attr person = Person()

    Container:
        Field:
            value >> person.name
```

Attribute Binding - Delegation

- :=
- Bi-directional
- Pushes and pulls values
- RHS must be an assignable expression

```
enamldef Main(Window):
    attr person = Person()

    Container:
        Field:
            value >> person.name
        Field:
            value := person.name
```

Attribute Binding - Notification

- `::`
- Right associative
- Executes code on change
- RHS can be any arbitrary Python code except for `def`, `class`, `return`, `yield`

```
enamldef Main(Window):
    attr person = Person()

    Container:
        Field:
            value >> person.name
            value :: print 'simple statement'
            value ::
                for i in range(10):
                    print i, person.name
```

Attribute Binding - Dependencies

- Enaml introspecting operators are extremely robust
- They can track almost any dependency in an expression
- It's all automatic!

```
Field:
    id: boss_info
    value << boss(school.room[class_id].teacher.id).in
```

```
Field:
    value << ', '.join([person.name for person in peop
```

Attribute Binding - Dependencies

- Enaml introspecting operators are extremely robust
- They can track almost any dependency in an expression
- It's all automatic!

```
Field:
    id: boss_info
    value << boss(school.room[class_id].teacher.id).in

Field:
    value << ', '.join([person.name for person in peop
```

Layout System

- Layout systems in GUI toolkits typically fall into 2 categories
  1. They don't exist and the developer is responsible for laying out widgets
  2. They use some form of nested box model
- #2 is preferable, but nested box models can be painful (Can we do a button ring?)
- We can do better!

Layout System - Constraints

- Enaml uses a constraints based layout system
- Constraints are specified as symbolic linear expressions of components
- This allows the convenience and ease of nested box models, but also the power and flexibility of manual layout

Layout System - Constraints

- Internally, uses the Cassowary linear constraint solver to do the heavy lifting in C++
  - OSX 10.7 now uses the same library
- Enaml provides convenience factories for auto generating constraints for most common cases.
- Layouts which are typically not possible are made possible

async UIs

DEMO

Thank You!