SCIENTIFIC APP BUILDING (WITH ENTHOUGHT TOOL SUITE)

Python Lunch and Learn series   26 MAY 2021

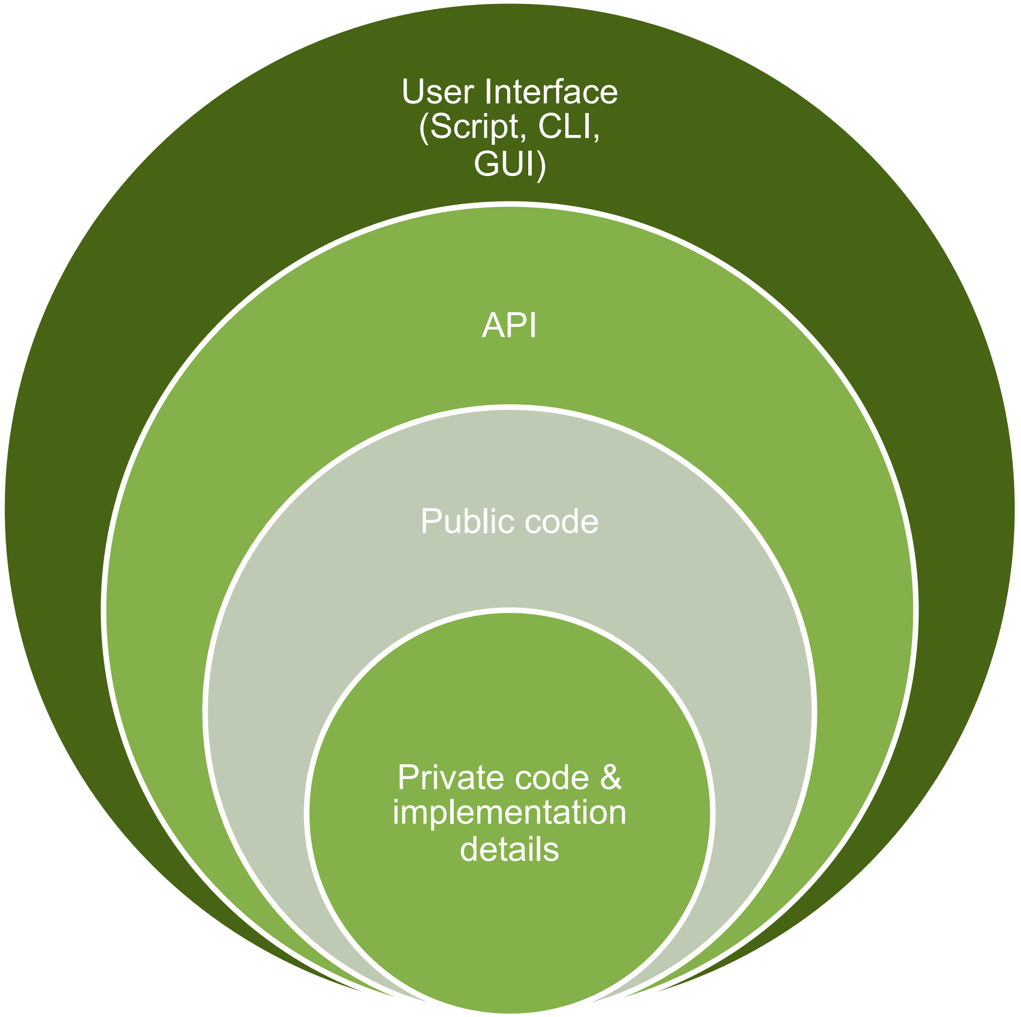
Jonathan Rocher,
Principal Software Architect,
KBI-Biopharma Inc.

Outline

1. The application stack: standing on the shoulders of giants
2. Introduction to ETS: traits
3. Introduction to ETS: traitsUI
4. Introduction to ETS: chaco
5. Advanced ETS: pyface
6. Design patterns with ETS: traits‘ adaptation, ...
7. Other packages: scimath

Foreword

Application design: layered package design

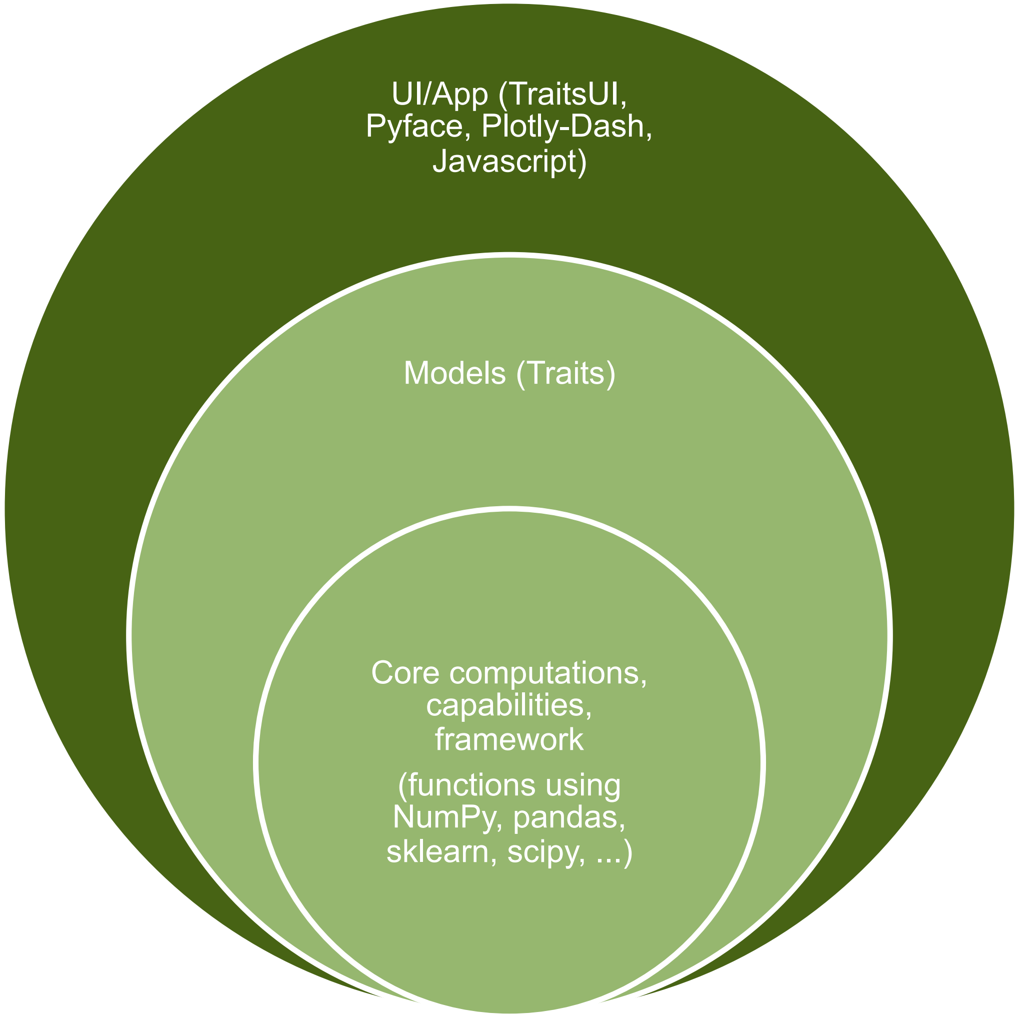

User Interface (Script, CLI, GUI)
API
Public code
Private code & implementation details

Layered package design for readability

To design your package, think about readability for someone else or for your future self. What do they/you need to get started? They/you need ways to use your package even if they don’t understand all the implementation details. Therefore:

1. Layer 1: the GUI. Include a documentation/README to describe how to use the UI.
2. Layer 2: the scripting layer. Include example scripts in the `scripts/` folder.
3. Layer 3: the rest of the API. `api.py` modules in sub-packages to know what they entry points are to invent new scripts.
4. Layer 4: the rest of the code. What modules contain data containers? What modules are just utility stuff readers can ignore at first? What modules do the heavy lifting of computing stuff. What is about UI stuff?
   ```
   pkg_name/app/
            model/
            view/
            io/
            compute/
            utils/
   ```
5. Layer 5: Inside modules, select functions or classes might be marked as private by prepending _ to their name.

Application design: layered package design


UI/App (TraitsUI, Pyface, Plotly-Dash, Javascript)
Models (Traits)
Core computations, capabilities, framework
(functions using NumPy, pandas, sklearn, scipy, ...)

KBI BIOPHARMA

Application Structure

GUI Application project structure

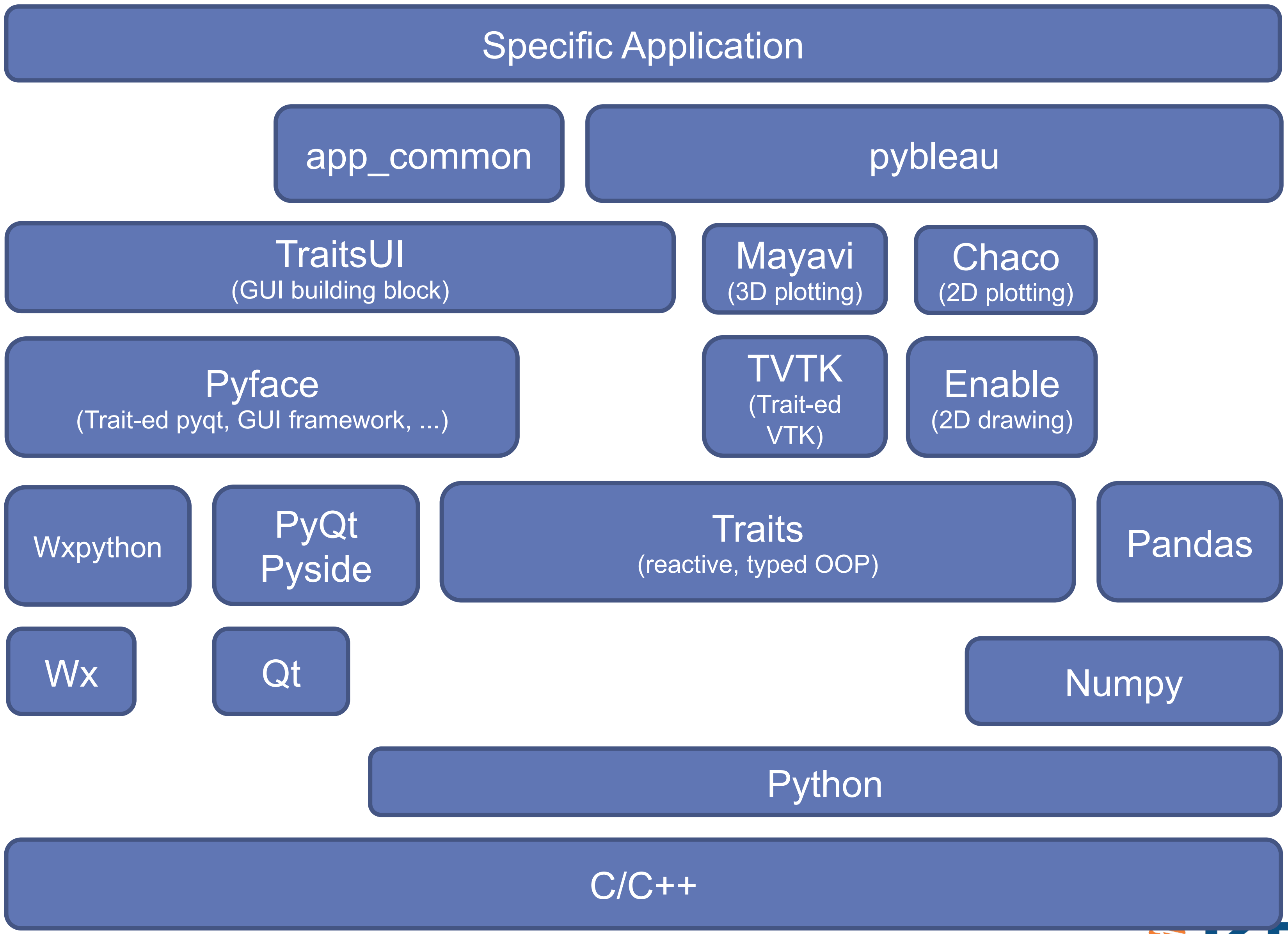
```
README.md
setup.py
scripts/
docs/
ci/__main__.py
   ...
pkg_name/app/
         model/
         ui/
         io/
         tools/
         utils/
         __init__.py
```

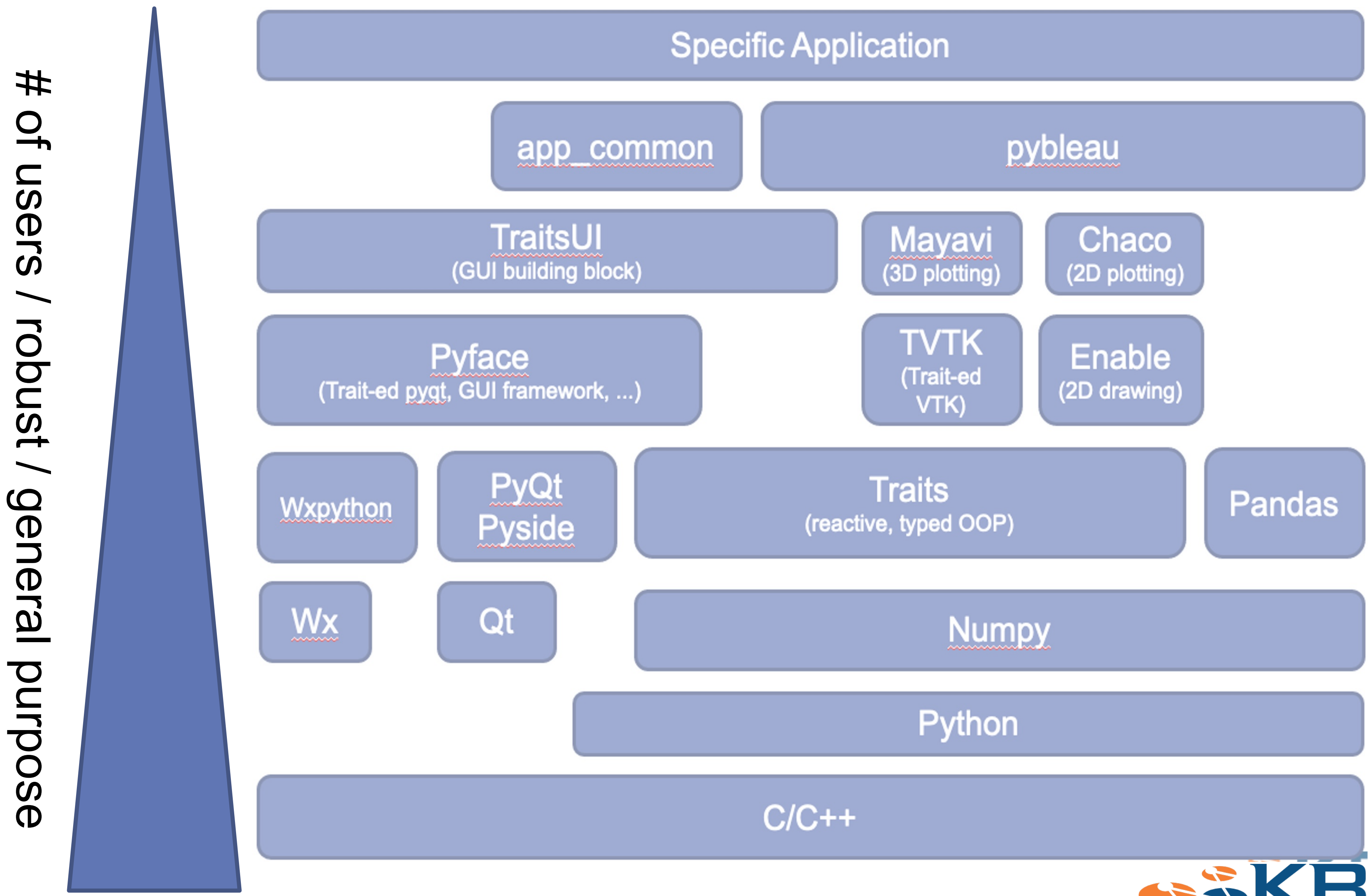
In each sub-package:

```
__init__.py
api.py
mod1.py
tests/test_mod1.py
```

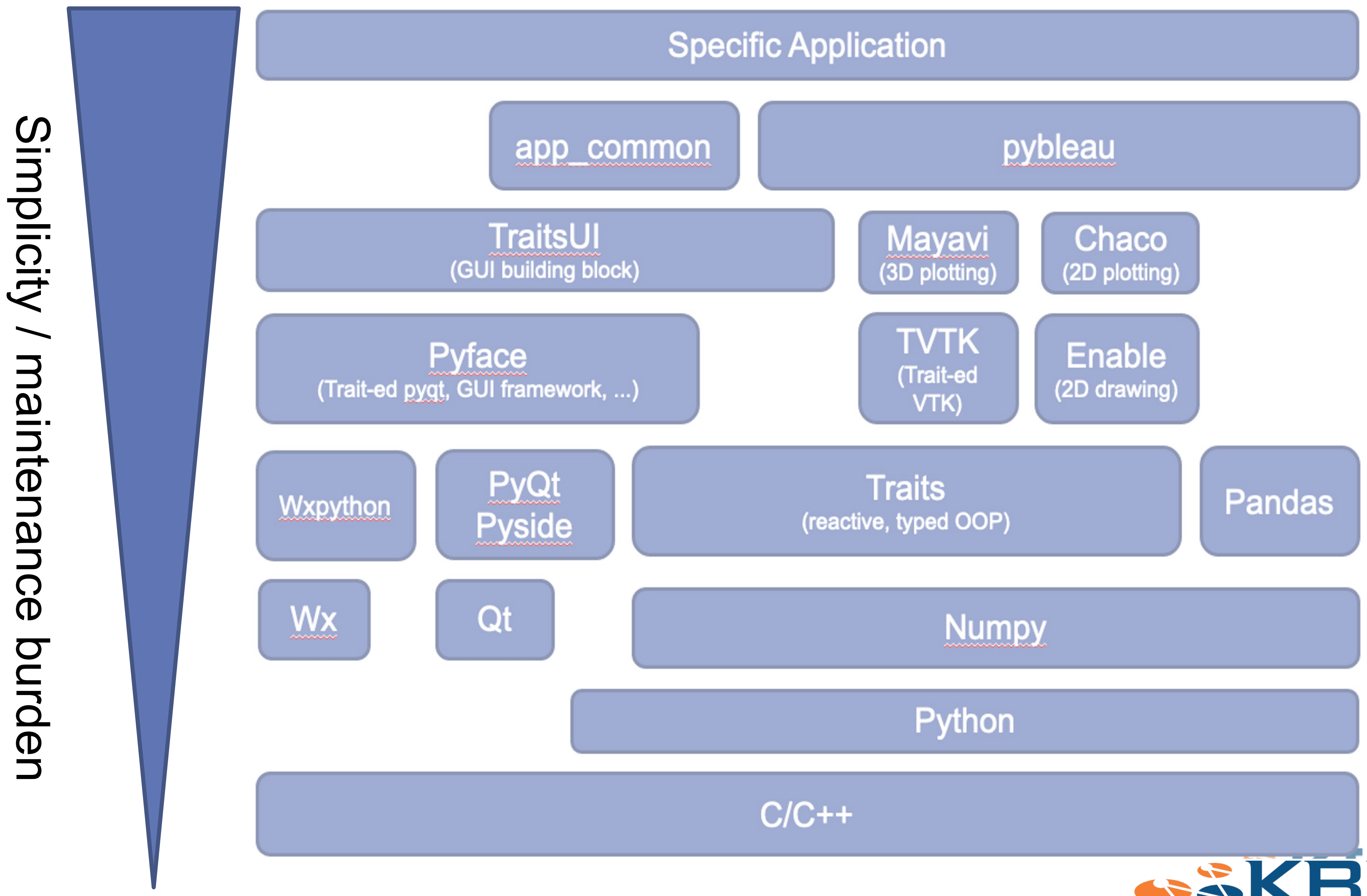
Application stack: what should live where?

KBI
BIOPHARMA


Specific Application
app_common
pybleau
TraitsUI
(GUI building block)
Mayavi
(3D plotting)
Chaco
(2D plotting)
Pyface
(Trait-ed pyqt, GUI framework, ...)
TVTK
(Trait-ed VTK)
Enable
(2D drawing)
Wxpython
PyQt
Pyside
Traits
(reactive, typed OOP)
Pandas
Wx
Qt
Numpy
Python
C/C++



# of users / robust / general purpose
Specific Application
app_common
pybleau
TraitsUI
(GUI building block)
Mayavi
(3D plotting)
Chaco
(2D plotting)
Pyface
(Trait-ed pyqt, GUI framework, ...)
TVTK
(Trait-ed VTK)
Enable
(2D drawing)
Wxpython
PyQt
Pyside
Traits
(reactive, typed OOP)
Pandas
Wx
Qt
Numpy
Python
C/C++


Simplicity / maintenance burden
Specific Application
app_common
pybleau
TraitsUI
(GUI building block)
Mayavi
(3D plotting)
Chaco
(2D plotting)
Pyface
(Trait-ed pyqt, GUI framework, ...)
TVTK
(Trait-ed VTK)
Enable
(2D drawing)
Wxpython
PyQt
Pyside
Traits
(reactive, typed OOP)
Pandas
Wx
Qt
Numpy
Python
C/C++

Guiding principles

1. Top layer is only useful for the 1 application built: represents the maintenance burden, and most fragile because only used by that 1 application/group.
2. `app_common` and `pybleau` developed at KBI, but used by 3 applications and counting. Also, open sourced, so can be tested and improved by outside users.
3. ETS used by dozens of applications developed at KBI, Enthought, JSR, Airbus, Exxon Mobil, Shell, Procter and Gamble, ... Provides a coherent and battle tested set of tools for our applications: simple UI (R&D grade).
4. The scipy ecosystem is used by tens of thousands of developers every day, supports near infinite algo complexity.
5. The Python language is used by millions of people everyday: it‘s the most robust part of the stack.

Intro to the Enthought Tool Suite

https://docs.enthought.com/ets/

What is ETS? Why ETS?

1. ETS is an open-source set of tools for (R&D-grade) scientific desktop (rich client) application, developed and maintained by Enthought Inc.
2. It leverages lower level GUI frameworks like Qt or Wx, but provides a simplified and unified interface to either suitable for simple GUI tools.
3. Additionally, it provides a 2D plotting framework, a 3D plotting framework (built on top of Kitware's VTK) and a few other more specialized capabilities (canvases, computation graph analysis, unit system, ...) that integrate easily with the suite.
4. Finally, it provides 3 levels of application frameworks, each capable of embedding the previous one, for smooth scalability. These are stored in 3 of the suite's packages: `traitsUI` (simplest), `pyface` (more scalable), `envisage` (plugin based).

https://docs.enthought.com/ets/

KBI BIOPHARMA

Intro to ETS: Traits

https://docs.enthought.com/traits/

What is Traits? Why Traits?

1. Traits is the core package in the Enthought Tool Suite.
2. It extends Python‘s OOP to implement 2 major functionalities:
    1. Type-aware classes compared to regular Python (and more generally more rigid OOP)
    2. Reactive programming: no matter how, if something changes, trigger a callback.
3. Both are valuable for building GUIs but `Traits` is valuable for headless tools too. The rigidity helps avoid mistakes while developing and the listener pattern avoid „spaguetti code“.
4. It works with TraitsUI to expose the data as a GUI window as simply as `obj.configure_traits()`.
5. The rest of ETS adds 2D and 3D plotting, several app frameworks and utilities to build on top of Traits objects.

Standard Python classes

Note: refer to refactor L&L deck for intro to OOP

```
import os

class MFIFileRepository:
   def __init__(self, url, name=""):
      self.name = name
      self.url = url
      self.exp_list = []
      self.scanned = False
      self.num_exp = 0

   def scan(self):
      self.exp_list = os.listdir(self.url)
      self.scanned = True

   def export(self, to_file="exp_data.h5"):
      ...
```

Automatically called upon creation

When we use the class (MFIFileRepository) to create one of these “objects”, it is called an instance (repo):

```
>>> repo = MFIFileRepository(url=".")
>>> repo.scan()
>>> print(repo.exp_list)
>>> repo.uel = "test"
```

KBI BIOPHARMA

Traits version

HasTraits classes (using the traits package) use a different provide 4 major improvements over standard classes: automatic initialization, type checking, listeners and automatic UI building.

```
import os
from traits.api import Bool, HasStrictTraits, Int, List, Str

class MFIFileRepository(HasStrictTraits):
   name = Str
   url = Str
   exp_list = List
   scanned = Bool
   num_exp = Int

   def scan(self):
      self.exp_list = os.listdir(self.url)
      self.scanned = True

   def export(self, to_file="exp_data.h5"):
      ...

>>> repo = MFIFileRepository(url=".")
>>> print(repo.scanned)
False
>>> repo.scan()
>>> print(repo.scanned)
True
>>> print(repo.exp_list)
>>> repo.export()
>>> repo2 = MFIFileRepository(url=3)
>>> repo.scaned = True # This will fail because of the typo!
```

Traits' listeners (the dynamic form)

```
import os
from traits.api import Directory, HasStrictTraits, Int,\
    List, observe, Str

class MFIFileRepository(HasStrictTraits):
    ...

    @observe("url")
    def scan(self):
        self.exp_list = os.listdir(self.url)
        self.scanned = True

    @observe("exp_list[]")
    def update_count(self):
        self.num_exp = len(self.exp_list)
```

Traits listeners: the static form (*)

```
import os
from traits.api import Directory, HasStrictTraits, Int, List,\
    on_trait_change, Str

class MFIFileRepository(HasStrictTraits):
   ...

   def _url_changed(self, obj, name, old, new):
      if new != "":
         self.scan()

   def scan(self):
      self.exp_list = os.listdir(self.url)
      self.scanned = True

   ...
```

https://docs.enthought.com/traits/traits_user_manual/notification.html

Traits' properties

Certain quantities or attributes may need to be built from custom code based on the value of other (changing) quantities but may be seldom useful. In that case, they may be computed lazily only upon request.

```
from traits.api import Directory, HasStrictTraits, Int,\
    List, observe, Property, Str

class MFIFileRepository(HasStrictTraits):
   exp_list = List
   num_exp = Property   # or Property(Int)

   @observe("url")
   def scan(self):
      self.exp_list = os.listdir(self.url)
      self.scanned = True

   def _get_num_exp(self):
      print("Computing the number of experiments...")
      return len(self.exp_list)

>>> repo = MFIFileRepository(url=".")
# scanning happens
>>> repo.exp_list
...
>>> repo.num_exp
Computing the number of experiments...
15
```

Traits' properties cont'd

Property can also be restricted to a certain type to allow type checking:

```
class MFIFileRepository(HasStricTraits):
   num_exp = Property(Int)
```

They can also explicitly specify what variable(s) they depend on:

```
class MFIFileRepository(HasStricTraits):
   num_exp = Property(Int, depends_on="exp_list[]")
```

Specifying the dependence(s) actually makes the property not lazy anymore, something that's desirable if the property's value is displayed in a UI, and must therefore update automatically:

```
>>> repo = MFIFileRepository()
>>> repo.url="."
# scanning happens
Computing the number of experiments...
>>> repo.num_exp
15
```

Note: If a property depends on more than 1 attribute, a single string must be specified, with attribute names separated by commas.

Note: If a property is to support value assignment, a special method must be created, with the name and signature following this pattern:

```
def _set_num_exp(self, value):
   # Custom code doing something with value
```

Then, something like `repo.num_exp = 4` becomes a supported statement.

The __init__() method (*)

Classes deriving from `HasTraits` don't typically need to override the `__init__` method because the parent's class takes care automatically of assigning attributes from the provided values. In certain circumstances though, some transformations need to be done before or after Traits initialization. Then, the `__init__` can be overridden but it's critical to call the parent's implementation at some point. Failing to do that will result in inconsistent behavior such as issues triggering expected listeners or with properties.

```
import os
from traits.api import HasTraits, HasStrictTraits, Int, List, Str
class MFIFileRepository(HasStrictTraits):
   name = Str
   url = Str
   exp_list = List
   scanned = Bool
   num_exp = Int

   def __init__(self, **traits):
      <CUSTOM STUFF>
      super(MFIFileRepository, self).__init__(**traits)
      <CUSTOM STUFF>

   def export(self, to_file="exp_data.h5"):
      ...

>>> repo = MFIFileRepository(url=".")
>>> repo.scan()
>>> print(repo.exp_list)
>>> repo.export()
```

Nesting objects with `Instance` (*)

Why do we need an `Instance` trait? Allows to compose multiple `HasTraits` classes together for a cleaner architecture. `Instance` is a way to convert any class into a `TraitType` so initialization, validation and listening is enabled.

```
from traits.api import Instance
from mypkg.model.mfi_repository import MFIRepository

class MFIAnalysis(HasStrictTraits):
   repository = Instance(MFIRepository)
   size = Int

   def summarize(self):
      df = self.repository.scan()
      return df.summary()

   def _repository_changed(self):
      ...

>>> repo = MFIRepository()
>>> analysis = MFIAnalysis(repository=repo)
>>> analysis.summarize()
```

A couple notes about `Instance` (*)

A couple of things to note about Instances: the default value is `None`, so don't forget to initialize.

```
>>> analysis = MFIAnalysis()
>>> analysis.repository
None
```

Initialize with:

```
class MFIAnalysis(HasStrictTraits):
   ...
   def _repository_default(self):
      return MFIRepository()
```

or equivalently

```
class MFIAnalysis(HasStrictTraits):
   repository = Instance(MFIRepository, ())
```

Additionally, `Instance` can receive an importable path to a class rather than a class itself:

```
class MFIAnalysis(HasStrictTraits):
   repository = Instance("MFIRepository")
```

or

```
   repository = Instance("mypkg.model.mfi_repository.MFIRepository")
```

“Free” GUI for a traits model

```
import os
from traits.api import Directory, HasStrictTraits, Int, List, Str

class MFIFileRepository(HasStrictTraits):
    name = Str
    url = Directory
    exp_list = List
    scanned = Bool
    num_exp = Int

    @on_trait_change("url")
    def scan(self):
        self.exp_list = os.listdir(self.url)
        self.scanned = True

    def export(self):
        ...

>>> repo = MFIFileRepository(url=".")
>>> repo.configure_traits()
```

Instead of a basic string, just to get a cooler widget when TraitsUI makes an automatic UI.

Intro to ETS: TraitsUI

https://docs.enthought.com/traitsui/

What is TraitsUI? When TraitsUI?

1. TraitsUI is a simple GUI builder that works with Traits classes.
2. `configure_traits()` automatically builds a UI, and launches the GUI event loop to allow interactions.
3. But customized views can be built, by building a `View` object which contains `Item`s, optionally combined inside `Group`s, and returning it by the `traits_view()` method.
4. More controls are offered via the `Item`'s attributes such as its `editor`, `label`, `show_label`, `width`, `height`, ...
5. Under the covers, it defaults to Qt as the backend (via PyQt or Pyside) to do the UI painting and user interaction, though it can use Wx and other backends and exposes a simple API, which is sufficient for most R&D tools. Backend, and wrapper are controlled by the `ETS_TOOLKIT` and `QT_API` env variable.

Controlling the view content

```
import os
from traits.api import Button, Directory, \
   HasStrictTraits, Int, List, Str
from traitsui.api import Item, View

class MFIFileRepository(HasStricTraits):
    ...
    def traits_view(self):
        view = View(
            Item("name"),
            Item("url")
            Item("exp_list"),
        )
        return view
```

KBI BIOPHARMA

Add actions to the GUI

```
from traits.api import Button, Directory, HasStrictTraits,\
   Int, List, Str
...

class MFIFileRepository(HasStrictTraits):
   ...
   num_exp = Int
   scan_button = Button("scan!")

   def traits_view(self):
      view = View(
         Item("name"),
         Item("url")
         Item("exp_list"),
         Item("scan_button")
      )
      return view

   def _scan_button_fired(self):
      self.scan()
   ...

>>> repo = MFIFileRepository(url=".")
>>> repo.configure_traits()
```

Control the view layout with TraitsUI

```
import os
from traits.api import Button, Directory, HasStrictTraits, Int, List, Str
from traitsui.api import HGroup, Item, VGroup, View

class MFIFileRepository(HasStricTraits):
    ...
    def traits_view(self):
        view = View(
            VGroup(
                HGroup(
                    Item("name"),
                    Item("url")
                ),
                VGroup(
                    Item("exp_list"),
                    Item("scan_button")
                ),
            ),
            title= "MFI Repository manager"
        )
        return view
```

Warning: if you forget the first VGroup directly inside View, each group would be displayed as a separate tab.

Tabbed view, Spring and Item attributes

```
import os
from traits.api import Button, Directory, HasStricTraits, Int, List, Str
from traitsui.api import HGroup, Item, Spring, Tabbed, VGroup, View

class MFIFileRepository(HasStricTraits):
    ...
    def traits_view(self):
        view = View(
            Tabbed(
                HGroup(
                    Item("name", width=300),
                    Spring(),
                    Item("url", label="Repo URL"),
                    label="First tab"
                ),
                VGroup(
                    Item("exp_list", style="readonly"),
                    Item("scan_button", show_label=False)
                ),
            ),
            title="MFI Repository manager"
        )
        return view
```

Note: Technically, Tabbed wasn’t strickly necessary here, but better for being more explicit about how the view should look.

KBI BIOPHARMA

Static text, visibility controls, View attrs

```
import os
from traits.api import Button, Directory, HasStricTraits, Int, List, Str
from traitsui.api import HGroup, Item, Label, OKCancelButtons, Spring, Tabbed,
VGroup, View

class MFIFileRepository(HasStricTraits):
    ...
    def traits_view(self):
        view = View(
            Tabbed(
                HGroup(
                    Item("name", width=300),
                    Spring(),
                    Item("url", label="Repo URL")
                ),
                VGroup(
                    HGroup(Spring(), Label("No experiment found!", Spring(),
                           visible_when="len(exp_list)==0"))
                    Item("exp_list", style="readonly"),
                    Item("scan_button", show_label=False)
                ),
            ),
            title="MFI Repository manager",
            resizable=True, width=600, height=900, buttons=OKCancelButtons
        )
        return view
```

Adding an icon

```
import os
from traits.api import Button, Directory, HasStrictTraits, Int, List, Str
from traitsui.api import HGroup, Item, Label, Spring, Tabbed, VGroup, View
from pyface.image_resource import ImageResource

class MFIFileRepository(HasStrictTraits):
   ...
   def traits_view(self):
      view = View(
         Tabbed(
            HGroup(
               Item("name", width=300),
               Spring(),
               Item("url", label="Repo URL")
            ),
            VGroup(
               HGroup(Spring(), Label("No experiment found!", Spring(),
                      visible_when="len(exp_list)==0"))
               Item("exp_list", style="readonly"),
               Item("scan_button", show_label=False)
            ),
         ),
         title="MFI Repository manager", icon=ImageResource("ftv_icon.png"),
         resizable=True, width=600, height=900
      )
      return view
```

Rule: Icon file needs to be in a folder called `images` next to where the `ImageResource` instance is created.

Valuable attributes to explore

1. `Item` class:
   `editor`, `style`: control which widget is used
   `label`, `show_label`, `tooltip`: control the label (if any) and tooltip
   `width`, `height`: control the size of the widget
   `visible_when`, `enabled_when`: allow dynamic views with widgets appearing or being controllable based on boolean expressions.
2. `Group` class attributes are same as `Item` except `editor` and:
   `show_border`: for drawing the group as subpanel
3. `View` class:
   `width`, `height`, `resizable`: control the size of the window
   `title`, `icon`: title of the window
   `handler`, `keybindings`: control the behavior of the view
   `scrollable`: whether to force the container to expand or make the panel scrollable.
   `buttons`: list of buttons for the master view (OK, Cancel, ...)

MVC pattern: separating model and view

Anything that relates to storing and transforming information remains in the model class. Anything that relates to displaying information, and transforming it is moved to a separate View class.

QUIZ: In the class below, what is part of the model? What is part of the view?

```
class MFIFileRepository(HasStricTraits):
   name = Str
   url = Directory
   exp_list = List
   scanned = Bool
   num_exp = Int
   scan_button = Button("scan!")

   def _scan_button_fired(self):
      self.scan()

   def scan(self):
      ...

   def export(self):
      ...

   def traits_view(self):
      ...
```

The Model:

```
import os
from traits.api import Bool, Directory, HasStrictTraits, \
   Int, List, Str

class MFIFileRepository(HasStrictTraits):
   name = Str
   url = Directory
   exp_list = List
   scanned = Bool
   num_exp = Int

   def scan(self):
      ...

   def export(self):
      ...
```

The View

```
from traits.api import Button, Instance
from traitsui.api import Item, ModelView, View

class MFIFileRepositoryView(ModelView):
   model = Instance(MFIFileRepository)
   scan_button = Button("scan!")

   def traits_view(self):
      view = View(
         Item("model.name"),
         Item("model.url", label="Repo URL"),
         Item("model.exp_list", style="readonly"),
         Item("scan_button", show_label=False),
         title= "MFI Repository manager",
         resizable=True
      )
      return view

    def _scan_button_fired(self):
      self.model.scan()
```

```
>>> model = MFIFileRepository(url="path")
>>> view = MFIFileRepositoryView(model=model)
>>> view.configure_traits()
```

Launching pop up views

```
from traits.api import Button, HasStrictTraits, Instance
from traitsui.api import Item, ModelView, OKCancelButtons, View

class MFIFileRepositoryView(ModelView):
   model = Instance(MFIFileRepository)
   scan_button = Button("scan!")

   ...

    def _scan_button_fired(self):
      popup = Popup()
      ui = popup.edit_traits(kind="livemodal")
      if ui.result:
         self.model.scan(fast=popup.fast_scan)

class Popup(HasStrictTraits):
    fast_scan = Bool
    view = View(Item("fast_scan"), buttons=OKCancelButtons)
```

As opposed to `configure_traits`, `edit_traits` should be used when wanting to open a new window when the GUI event loop is already running.

The `kind` attribute controls the behavior of the new window: `modal` means that it is blocking. `live` means that no object copy is done.

Exercise: how to improve the popup UI to provide more guidance to the user?

Launching pop up views

```
from traits.api import Button, HasStrictTraits, Instance
from traitsui.api import Item, ModelView, OKCancelButtons, View

class MFIFileRepositoryView(ModelView):
   model = Instance(MFIFileRepository)
   scan_button = Button("scan!")

   ...

    def _scan_button_fired(self):
      popup = Popup()
      ui = popup.edit_traits(kind="livemodal")
      if ui.result:
         self.model.scan(fast=popup.fast_scan)

class Popup(HasStrictTraits):
    fast_scan = Bool
    view = View(Item("fast_scan", label="Fast scan?"),
                buttons=OKCancelButtons,
                title="Select scan type")
```

Advanced TraitsUI

The Controller

```
from traits.api import Button, Instance
from traitsui.api import Item, ModelView, View, Handler, Controller

class MFIFileRepositoryView(ModelView):
   model = Instance(MFIFileRepository)
   scan_button = Button("scan!")

   def traits_view(self):
      view = View(
         Item("model.name"),
         Item("model.url", label="Repo URL"),
         Item("model.exp_list", style="readonly"),
         Item("scan_button", show_label=False),
         title= "MFI Repository manager",
         resizable=True
         buttons=[...]
         handler=MFIFileRepositoryHandler()
      )
      return view

    def _scan_button_fired(self):
      self.model.scan()
```

```
>>> model = MFIFileRepository(url="path")
>>> view = MFIFileRepositoryView(model=model)
>>> view.configure_traits()
```

Nesting views with the `InstanceEditor`

A key to scalable applications is the ability to build views from view components (or models from other models). Embedding views inside views can be done by building multiple `HasTraits` classes, each with their own views, then compose them in a top level view.

```
class MFIApp(HasStrictTraits)
   repository = Instance(MFIFileRepositoryView)
   ...
   view = View(
       Item("repository", editor=InstanceEditor(), style="custom"),
       ...
       title="MFI App"
   )


class MFIFileRepositoryView(ModelView):
   model = ...
   launch_button = ...
   def traits_view(self):
      return View(
         Item("model", ...), Item("launch_button", ...)
      )
```

Tabular data: so many editors...

Tabular data is so common in science that `TraitsUI` offers many options to display data in a table. Here is a (semi-)complete list:

- `ArrayEditor` (default for an `Array` trait)
- `DataFrameEditor`
- `TableEditor`
- `TabularEditor`

The first 2 editors are designed to offer a quick (automatic) way to display a `NumPy` array or a `Pandas’ DataFrame`.

The last 2 editors are designed to offer a flexible way to display data in a tabular form even when the data is stored in non-tabular objects such as a list of python objects. That flexibility is enabled by having an Adapter between the model and the editor to do the translation.

ArrayEditor & DataframeEditor

```
from traits.api import Array, Instance
from traitsui.api import ArrayEditor, Item, View
from traitsui.ui_editors.data_frame_editor import DataFrameEditor

class Test(HasStrictTraits):
   raw_data = Array
   df = Instance(DataFrame)
   def traits_view(self):
      editor_kw = dict(show_index=True, columns=[...],
                       fonts=..., formats=...)
      data_editor = DataFrameEditor(selected_row="selected_idx",
                                    multi_select=True, **editor_kw)
      return View(
         Item("raw_data", editor=ArrayEditor(), label="numpy array"),
         Item("df", editor=data_editor, label="Pandas DF"),
      )
```

Custom table with a `TabularEditor`

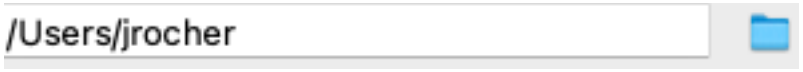
```
from traits.api import Button, Instance
from traitsui.api import Action, Handler, Item, ModelView, OKButton, View
```

Custom table with a `TableEditor`

```
from traits.api import Button, Instance
from traitsui.api import Action, Handler, Item, ModelView, OKButton, View
```

Other `Editors` worth knowing about

- `FileEditor` is the default editor for a `File` trait. The `simple` mode is a file selector



  but the custom style builds a full featured tree navigator.
- `TreeEditor` allow you to navigate any object as a tree, not just files.

Menus and toolbars

It can be done in pure TraitsUI though I recommend to use the pyface framework if you need that. See pyface section

Intro to ETS: Chaco

https://docs.enthought.com/chaco/

What is Chaco? When use Chaco?

1. Chaco is a 2D plotting library designed to work with the rest of the ETS to embed plots into scientific desktop applications.
2. It is powerful, well-architected, designed to allow a wide variety of custom interactions. It is currently badly documented, and therefore hard to learn on your own. In that sense, it is distinct in its purpose from `Matplotlib` (static plots) or `plotly` (interactive web-based plots).
3. The core concepts when building a chaco plot are:
   - The `ArrayPlotData` designed to hold data, and watch for data changes.
   - The `Plot` and other plot containers which are designed to hold all the components of a plot.
   - The renderers which are representations of the data.
4. Chaco plots are a specialized type of an `enable`'s `Component`, the underlaying 2D drawing library.

Chaco demos

Demos illustrates a wide array of tools and interactions to explore data (clone `chaco` from github.com, go to `examples/demo/` folder):

- `basic/scatter_inspector.py`
- `basic/image_inspector.py`
- `basic/cmap_scatter.py`
- `basic/contour_cmap_plot.py`
- `basic/inset_plot.py`
- `xray_plot.py`
- `zoomed_plot/zoom_plot.py`
- `data_labels.py`
- `advanced/spectrum.py`

See `demo.py` for a way to browse many more demos...

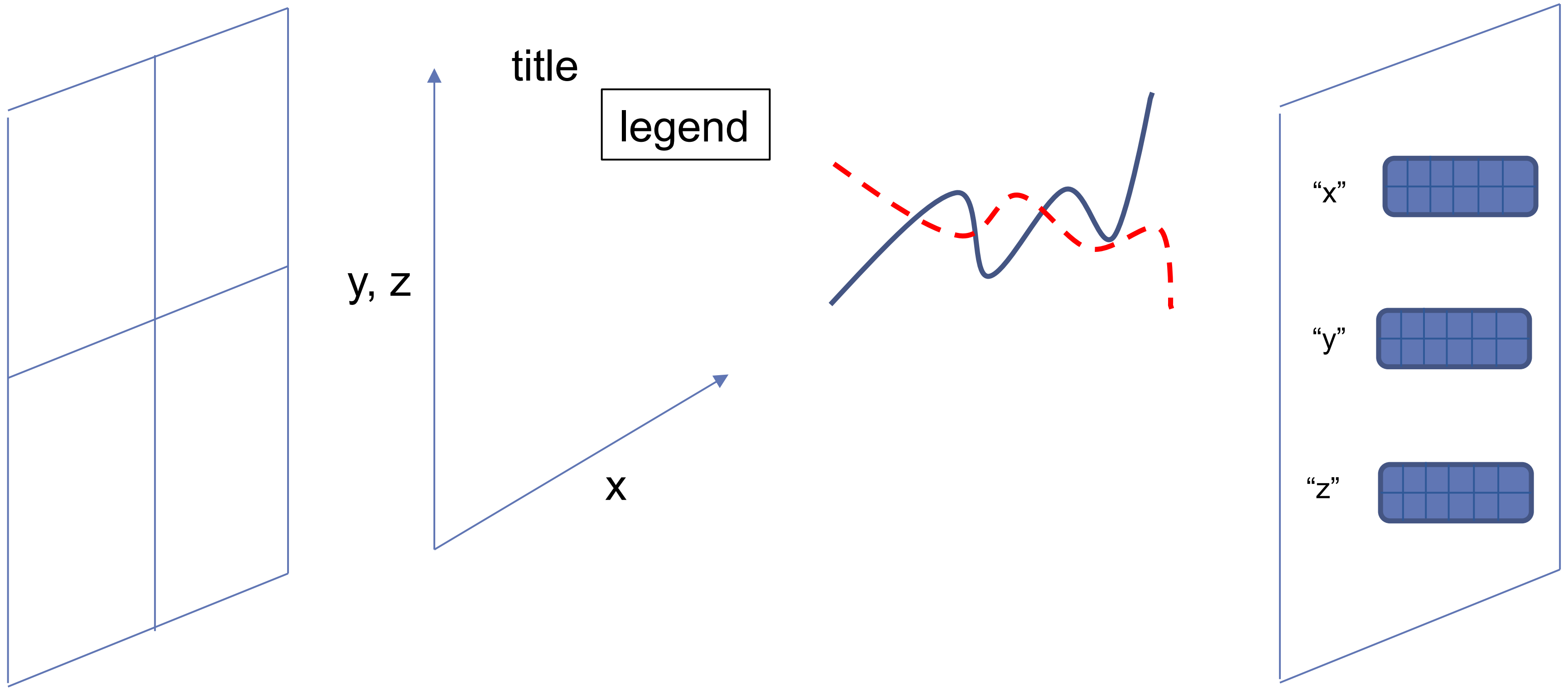
Chaco plot architecture

Containers (`HPlotContainer`, `VPlotContainer`)

Renderer container (`Plot`)

Renderer(s) (`LinePlot`, `ScatterPlot`, ...)

Data (`ArrayPlotData`)



First plot

```
from numpy import array, linspace, sin, cos, pi
...
from chaco.api import ArrayPlotData, Plot
from enable.api import ComponentEditor


class SimpleApp(HasStrictTraits):
    plot = Instance(Plot)

    view = View(Item("plot", editor=ComponentEditor()),
                title="Hello world", resizable=True)

    def _plot_default(self):
        x = linspace(0, 2*pi, 100)
        y = sin(x)
        y2 = cos(x)
        array_data = ArrayPlotData(time=x, data=y, data2=y2)
        plot = Plot(array_data)
        plot.plot(("time", "data"))
        plot.plot(("time", "data2"))
        return plot
```

To run it:

```
app = SimpleApp()
app.configure_traits()
```

Controlling plot/renderer/item attributes

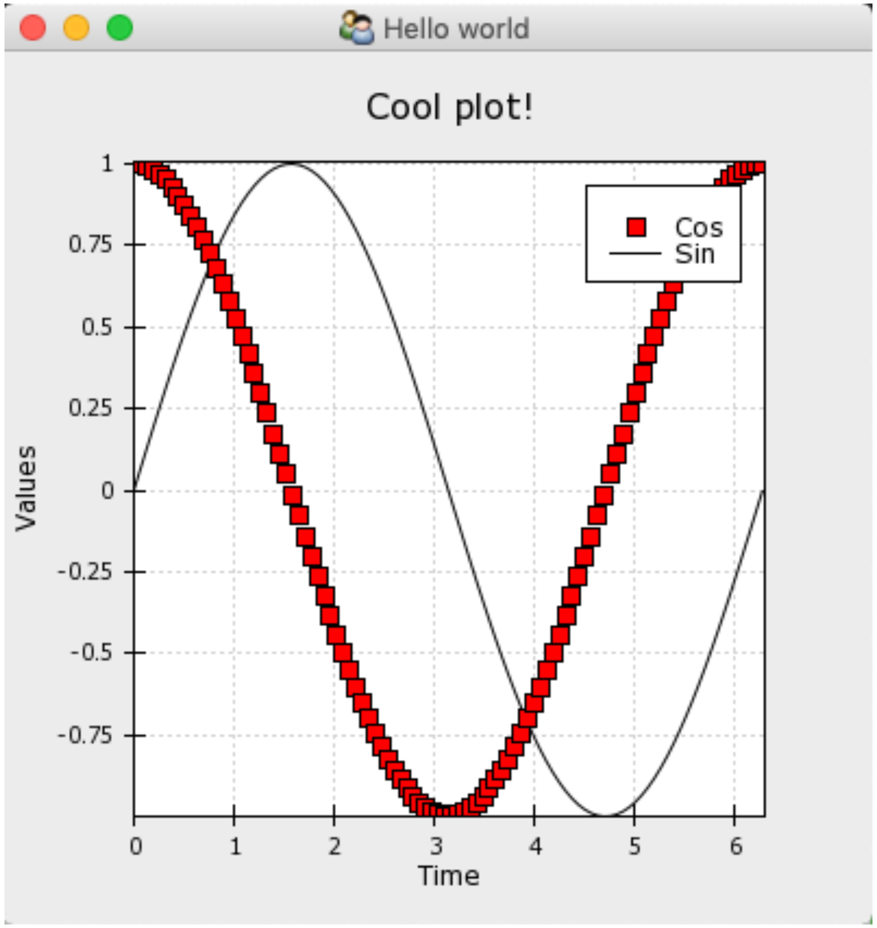
```
from numpy import array, linspace, sin, cos, pi
...
class SimpleApp(HasStrictTraits):
    plot = Instance(Plot)

    view = View(Item("plot", editor=ComponentEditor(), show_label=False),
                title="Hello world", resizable=True)

    def _plot_default(self):
        ...
        plot.plot(("time", "data"), type="line", name="Sin")
        plot.plot(("time", "data2"), type="scatter", color="red",
                  name="Cos")
        plot.title = "Cool plot!"
        plot.legend.visible = True
        plot.x_axis.title = "Time"
        plot.y_axis.title = "Values"
        plot.padding_left = 60
        return plot
```

Exercise: Explore the `Plot` class and its parents to review their attributes and the things that can be controlled.

Adding simple tools

```
...
from chaco.tools.api import PanTool, ZoomTool

class SimpleApp(HasStrictTraits):
    ...
    def _plot_default(self):
        ...
        plot.plot(("time", "data"), type="line", name="Sin")
        plot.plot(("time", "data2"), type="scatter", name="Cos")
        ...
        pan_tool = PanTool(component=plot)
        zoom_tool = ZoomTool(component=plot)
        plot.tools.append(pan_tool)
        plot.tools.append(zoom_tool)
        return plot
```

Adding simple tools



2 plots, side by side

```
from numpy import array, linspace, sin, cos, pi
from chaco.api import ..., HPlotContainer

class SimpleApp(HasStrictTraits):
    plot = Instance(HPlotContainer)

    view = View(Item("plot", editor=ComponentEditor(), show_label=False),
                title="Hello world", resizable=True)

    def _plot_default(self):
        ...
        plot = Plot(data)
        plot.plot(("time", "data"))
        plot2 = Plot(data)
        plot2.plot(("time", "data2))

        container = HPlotContainer()
        container.add(plot, plot2)
        return container
```

Synchronized plots

```
from numpy import array, linspace, sin, cos, pi
...

class SimpleApp(HasStrictTraits):
    plot = Instance(HPlotContainer)

    view = View(Item("plot", editor=ComponentEditor(), show_label=False),
                title="Hello world", resizable=True)

    def _plot_default(self):
        ...
        plot = Plot(data)
        plot.plot(("time", "data"))
        plot2 = Plot(data)
        plot2.plot(("time", "data2))

        plot.range2d = plot2.range2d
        container = HPlotContainer()
        container.add(plot, plot2)
        return container
```

Controlling the plot from UI

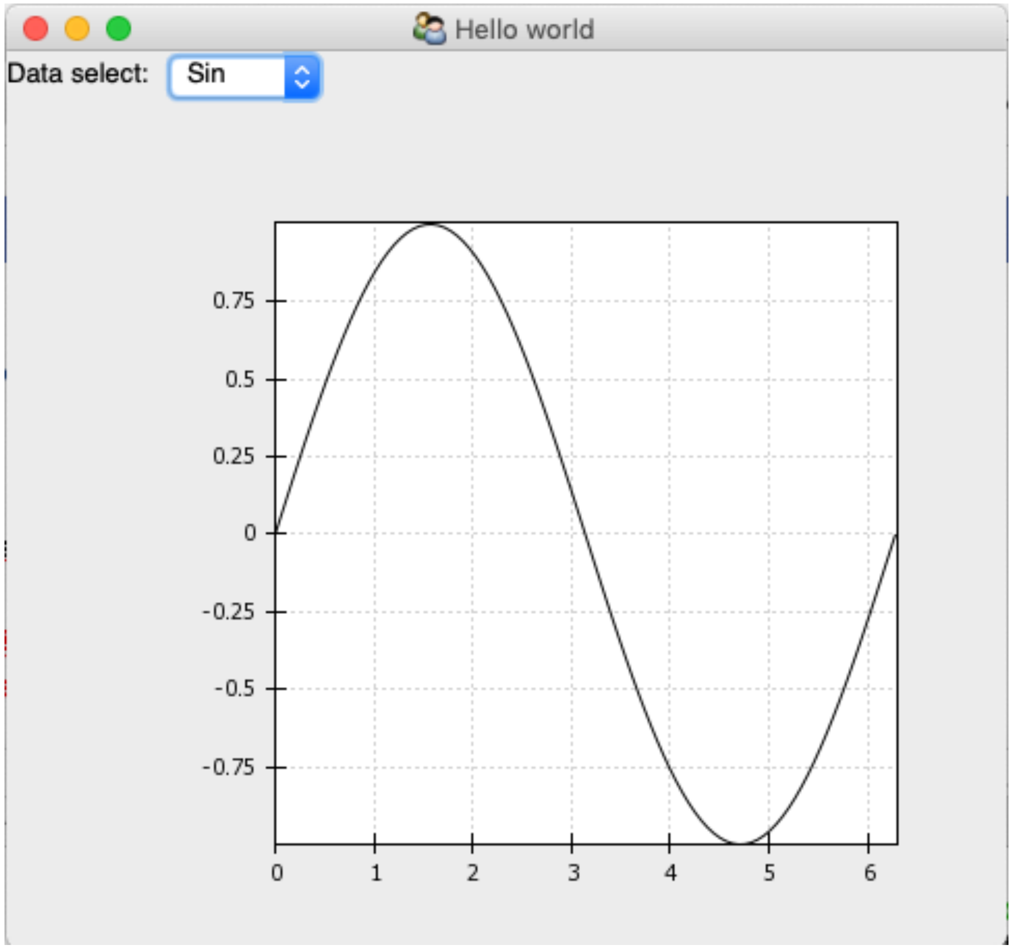
Info: the `ArrayPlotData` has a `set_data()` method to set a plotted dimension to a new array.

Exercise: Modify the current example to add a drop down widget to control whether to plot the sin or cos of the x values.

Controlling the plot from UI

```
...
class SimpleApp(HasStrictTraits):
    ...
    data_select = Enum(["Sin", "Cos"])
    data = Instance(ArrayPlotData)
    view = View(
        Item("data_select"),
        Item("plot", editor=ComponentEditor(), show_label=False),
        title="Hello world", resizable=True
    )
    def _plot_default(self):
        x = linspace(0, 2*pi, 100)
        y = sin(x)
        self.data = ArrayPlotData(time=x, data=y)
        plot = Plot(self.data)
        plot.plot(("time", "data"), type="line")
        return plot
    def _data_select_changed(self):
        x = linspace(0, 2*pi, 100)
        y = sin(x) if self.data_select == "Sin" else cos(x)
        self.data.set_data("data", y)
```

Controlling the plot from UI



Interactive plots: Chaco tools

Advanced ETS

ETS advanced functionalities

- What happened to the “C” in MVC? Controllers, custom buttons and key bindings.
- Interfaces and Adaptation patterns with Traits
- `pyface`’s Task framework for scalable GUI applications
- Customizing GUIs beyond `TraitsUI`’s capabilities

TraitsUI's Controller: custom button

```
from traits.api import Button, Instance
from traitsui.api import Action, Handler, Item, ModelView, OKButton, View

export_button = Action(name='Export', action="do_export")

class MFIFileRepositoryView(ModelView):
   ...
   def traits_view(self):
      view = View(
         ...
         buttons=[OKButton, export_button]
         handler=MFIFileRepositoryHandler()
      )
      return view

class MFIFileRepositoryHandler(Handler):
   def do_export(self, info):
      model = info.object.model
      model.to_preference_file()
```

Required signature for any handler method. `info` is a `UIInfo` object with a handle on the view object (called `object`) and the UI panel (called `ui`).

The Controller: key bindings

```
from traits.api import Button, Instance
from traitsui.api import Action, Handler, Item, ModelView, OKButton, View

export_button = Action(name='Export', action="do_export")

class MFIFileRepositoryView(ModelView):
   ...
   def traits_view(self):
      view = View(
         ...
         buttons=[OKButton, export_button]
         key_bindings=[KeyBinding(binding1='Ctrl-Right',
                                  description='Super cool binding',
                                  method_name='do_export')]
         handler=MFIFileRepositoryHandler()
      )
      return view

class MFIFileRepositoryHandler(Handler):
   def do_export(self, info):
      ...
```

Advanced Traits

Traits‘ adaptation

Adaptation is one of the most famous design pattern. It involves 3 types of objects:

1. Interfaces
2. Adapters
3. Objects (classes) that need to be adapted to an interface

Advanced ETS: pyface

https://docs.enthought.com/pyface/

What is pyface?

- `pyface` can globally be thought as a trait-ed version of pyside/pyqt. TraitsUI uses it to build views that are toolkit agnostic. It's also a grab bag of useful application building tools.
- `pyface` is badly documented ☹ and certain parts are deprecated (`workbench` for example).
- Valuable components of pyface for app development:
    1. Quick native dialogs (**`error, warning, information, confirm, ...`**)
    2. The task framework for mid-size application building
    3. Timer class for timed events (streaming data or updating UI in general)
    4. General GUI objects like clipboard, splash screen, about dialogs, ...

`pyface`'s dialogs

Good to know about the following dialogs to avoid having to use custom TraitsUI tools for these standard use cases. Refer to `pyface` documentation to learn more about these.

Informative dialogs:

- `error(parent, msg, title="Error")`
- `warning(parent, msg, title="Warning")`
- `information(parent, msg, title="Info")`

Usage:

```
error(None, "blah blah")
```

will automatically launch the dialog in modal way. (The parent of these dialogs can just be set to `None`.)

Basic question dialogs:

- `confirm()`
- `...`

Usage:

```
from pyface.api import confirm, YES, NO
response = confirm(None, "blah blah")
if response == YES:
    ...
```

Native file dialogs:

- `FileDialog()`
- `DirectoryDialog()`

Usage:

```
from pyface.api import FileDialog, OK
file_dialog = FileDialog(title='Export User Data',
                         action='save as',
                         wildcard="My Data (*.txt)|*.txt")
file_dialog.open()
if file_dialog.return_code == OK:
    path = file_dialog.path
    ...
```

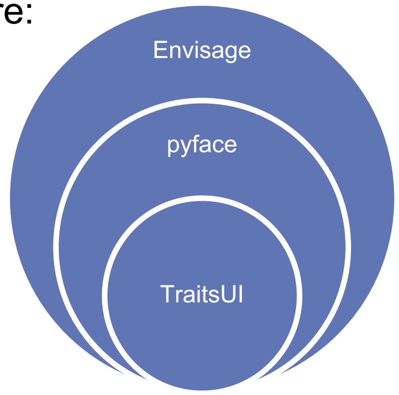
https://docs.enthought.com/pyface/

ETS's application frameworks

ETS provides 3 frameworks to build applications:

1. Pure TraitsUI: 1 master view, exposing menus and tools and built from one or more subviews (using the `InstanceEditor`). Recommended for **very small** applications, described in TraitsUI section before.
2. Pyface's `TaskApplication` adds multiple layers to the framework: embeds TraitsUI view elements in layers handling layout, menus, windowing system, event loop management, resource management. Recommended for **mid-size** applications, described in the pyface documentation and next few slides.
3. ETS' `envisage` adds a plugin system around all this (and its own Application object) to embed tasks into plugins reusable across applications. Envisage contains ready-to-use plugins for a few standard features. Recommended for **very large** applications, described in the envisage documentation: https://docs.enthought.com/envisage/ .

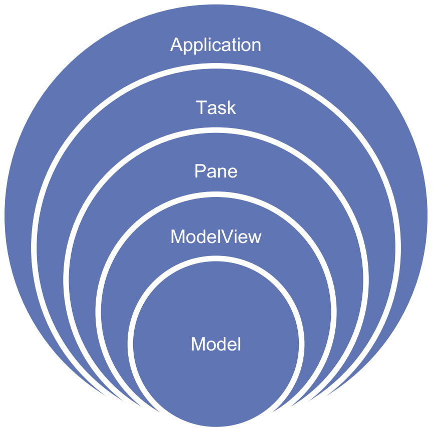
Good news: each layer embeds the previous one. So an application can grow from tiny to enormous, and most of the code is unchanged: it just gets embedded more:

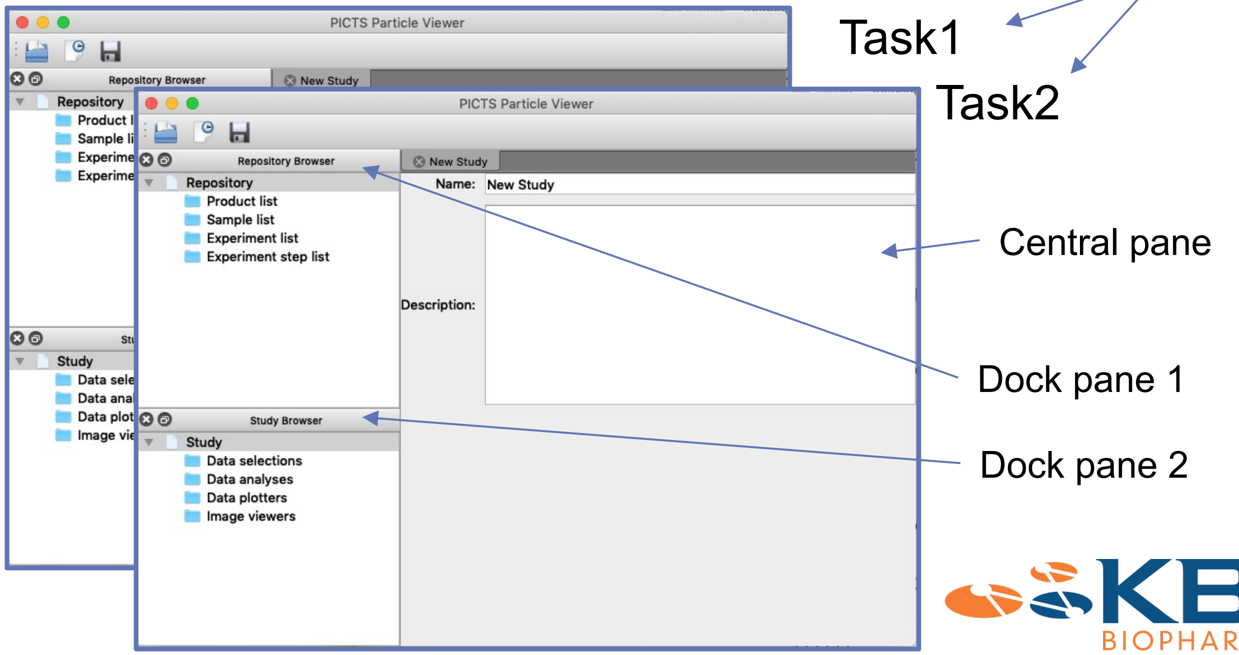


`pyface`'s Task application framework

This framework is used by many applications developed at KBI. It provides 4 layers of ownership and application development:

1. Application layer: only 1 instance, responsible for launching the GUI event loop, creating windows, initializing and cleaning up global resources (`pyface.tasks.TasksApplication`).
2. Task: owns single window, responsible for building window (made of multiple `TaskPane`), creating all the menu and toolbar entries (`pyface.tasks.task.Task`), and cross pane communication.
3. Pane layer: panel to be moved around in the Task, or shown/hidden, made of TraitsUI views (`pyface.tasks.task_pane.TaskPane`).
4. UI components (traitsUI): where the UI components are actually built.





Repository
- Product list
- Sample list
- Experiment list
- Experiment step list

Study
- Data selections
- Data analyses
- Data plotters
- Image viewers

Name: New Study

Description:

`pyface`'s Task app: hello world

We can start with the 2 most inner layers which we are now familiar with:

```
class HelloWorld(HasStrictTraits):
    content = Str("Hello world")

class HelloWorldView(ModelView):
    model = Instance(HelloWorld, ())
    def traits_view(self):
        return View(
            Item("model.content")
        )
```

At the application layer, we can just subclass TasksApplication and will need to provide the window building tools:

```
from pyface.tasks.api import TasksApplication, TaskFactory
class HelloWorldApp(TasksApplication):
    def _task_factories_default(self):
        return [TaskFactory(factory=HelloWorldTask)]
```

pyface's Task app: hello world

Then, per our layering drawing, we need to build the missing links between the application and the TraitsUI view:

```
from pyface.tasks.api import Task, TraitsTaskPane
class HelloWorldTask(Task):
    def create_central_pane(self):
        return HelloWorldPane()


class HelloWorldPane(TraitsTaskPane):
    pane_element = Instance(HelloWorldView, ())
    def traits_view(self):
        return View(
            Item("pane_element", editor=InstanceEditor(), style="custom")
        )
```

Finally, to run the application, just call its run method:

```
app = HelloWorldApp()
app.run()
```

Beyond “hello world”: tasks’ attr, methods

Understanding how to use this framework means understanding how to leverage the API for the 3 layers involved in the framework. The main methods and attributes to understand are the following.

Application’s interface:

- `start()`
- `create_task_window()`
- `close()`
- `tasks_created`
- `active_task`

Task’s interface:

- `create_central_pane()`
- `create_dock_panes()`
- `activated()`
- `prepare_destroy()`
- `menu_bar, tool_bars`
- `status_bar`
- `window` (for e.g. to control the window’s name)

Pane’s interface:

- `task` (to access the “parent”)
- `traits_view`() (to control how it renders)

Example: adding menu entries

Understanding how to use this framework means understanding how to leverage the API for the 3 layers involved in the framework. The main methods and attributes to understand are the following.

Example: custom actions upon window creation/destruction

Understanding how to use this framework means understanding how to leverage the API for the 3 layers involved in the framework. The main methods and attributes to understand are the following.

Beyond “hello world”: a good central pane

`pyface.tasks` comes with a useful general-purpose pane already implemented: the SplitEditorAreaPane.

```
from pyface.tasks.api import Editor, SplitEditorAreaPane, Task
class MyTask(Task):
    ...
    central_pane = Instance(SplitEditorAreaPane)

    def create_central_pane(self):
        self.central_pane = SplitEditorAreaPane()
        return self.central_pane
```

Opening objects in the pane requires to invoke its `edit` method, providing the object to open and an editor for it:

```
    def custom_method(self):
        obj = …
        self.central_pane.edit(obj, factory=MyEditor)
```

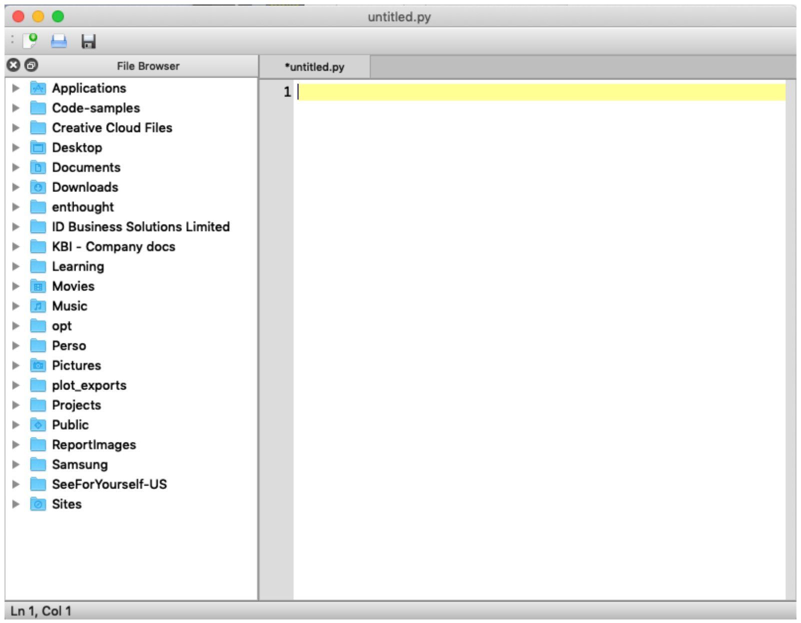
The editor/factory must be a class which sets the `control` attribute in the create method

```
class MyEditor(Editor):
    def create(self, parent)
        ui = self.obj.edit_traits(kind="subpanel", parent=parent)
        self.control = ui.control # set to the qt control
```

`pyface`'s Task application example

Just like any other ETS project, pyface's contains a lot of examples in its examples folder, in particular a full fledge Task application example:

https://github.com/enthought/pyface/blob/master/examples/application/python_editor/python_editor_application.py



Advanced ETS: fancier tools w/ Traits & TraitsUI

Traits(UI): going beyond defaults

Like any good tool in python, Traits/TraitsUI's classes offer customization through setting non-default values for (optional) attributes. This allows to customize tools beyond the defaults. For e.g., let's look at a basic tool like:

```
class SimpleTool(HasStrictTraits):
    text = Str
    view = View(Item("text"))
```

Implicitely, the view uses the default text editor in its "simple" form. That's equivalent to:

```
view = View(Item("text", editor=TextEditor(),
                 style="simple"))
```

Going beyond the default values means doing any of the following:

1. using a more precise (subclass) Trait,
2. setting some optional attributes of the Trait to control its (supported) values,
3. trying the "`custom`" style for the default editor to see what widget is used there,
4. setting some optional attributes of the TraitsUI editor to control its rendering,
5. trying a different editor explicitely.

Using a more precise Trait

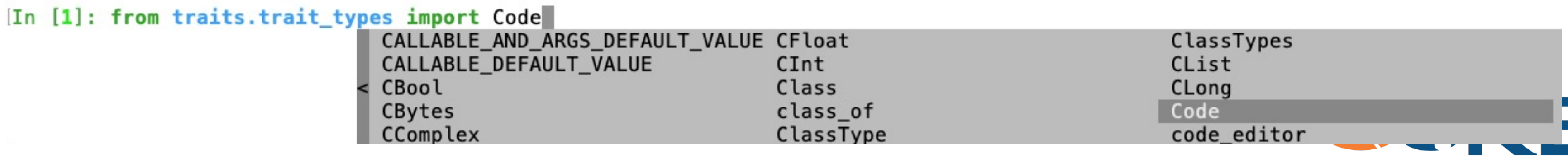
An easy way to have a more precise editor would be to use a different Trait class. For example, `File` or `Directory` subclass from `Str` and have different default editors (respectively `FileEditor` and `DirectoryEditor`):

```
class SimpleTool(HasStrictTraits):
    text = File
    view = View(Item("text"))
```

or equivalently:

```
class SimpleTool(HasStrictTraits):
    text = Str
    view = View(Item("text", editor=FileEditor(),
                     style="simple"))
```

To know all available trait types, you can look at the content of traits.trait_types using `IPython`:

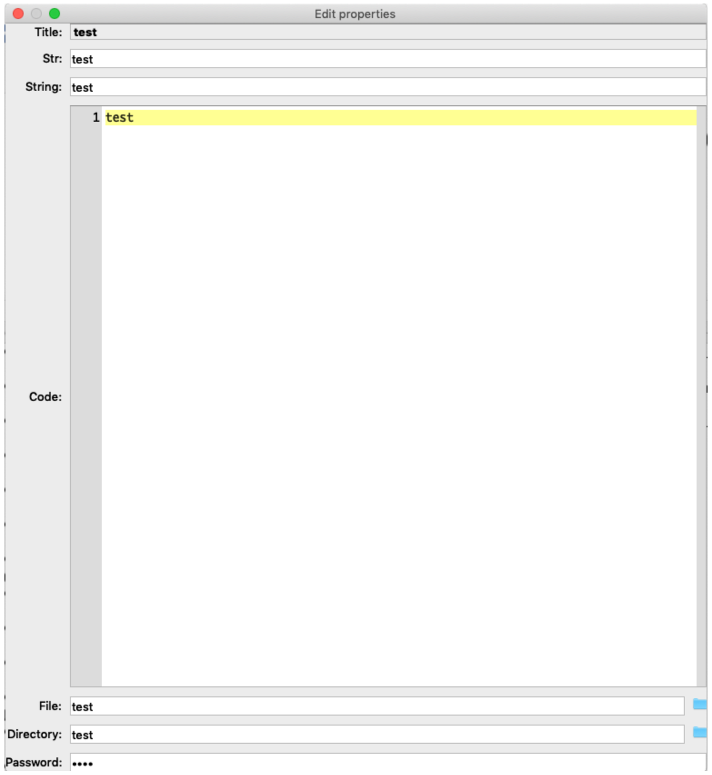


```
In [1]: from traits.trait_types import Code
                CALLABLE_AND_ARGS_DEFAULT_VALUE CFloat     ClassTypes
                CALLABLE_DEFAULT_VALUE          CInt       CList
               < CBool                          Class      CLong
                CBytes                          class_of   Code
                CComplex                        ClassType  code_editor
```

Using a more precise Trait

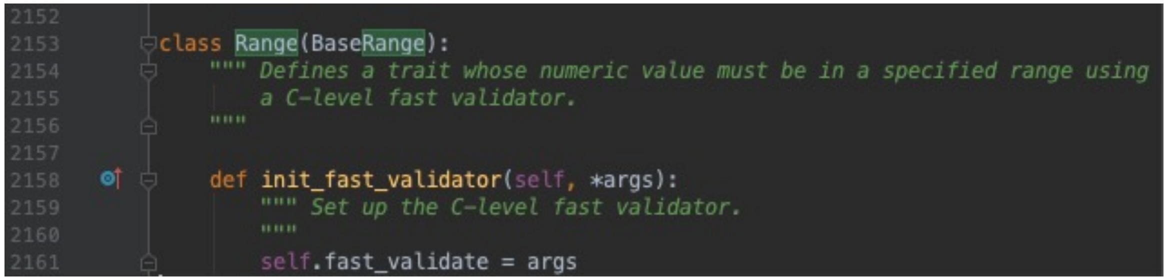
For e.g., the following Trait types all use a different editor/widget even though they all store a string:

```
from traits.api import HasStrictTraits, \
   Code, Directory, File, Password, \
   Str, String, Title

class SimpleTool(HasStrictTraits):
   str = Str("test")
   code = Code("test")
   password = Password("test")
   file = File("test")
   directory = Directory("test")
   string = String("test")
   title = Title("test")

   view = View(
      "title", "str", "string", "code",
      "file", "directory", "password"
   )
```



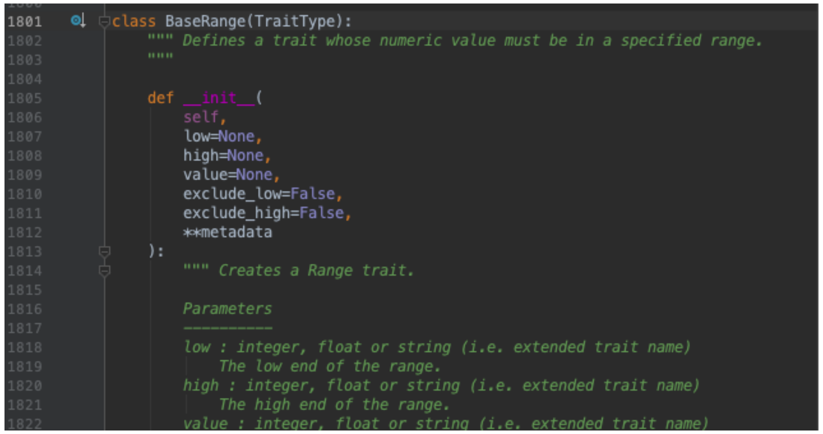
Optional Trait attributes

Another way to control behavior is to set optional attributes on the Trait used. For example, when using a `Range` trait, jumping (using a smart code editor) to the definition of that class provides quick access to the list of attributes that can be set:



```python

class Range(BaseRange):
    """ Defines a trait whose numeric value must be in a specified range using
        a C-level fast validator.
    """

    def init_fast_validator(self, *args):
        """ Set up the C-level fast validator.
        """
        self.fast_validate = args
```

There are no attributes directly on that class, but jumping to its base class `BaseRange` provides options to refine control by passing keyword args to the constructor:



```python
class BaseRange(TraitType):
    """ Defines a trait whose numeric value must be in a specified range.
    """

    def __init__(
        self,
        low=None,
        high=None,
        value=None,
        exclude_low=False,
        exclude_high=False,
        **metadata
    ):
        """ Creates a Range trait.

        Parameters
        ----------
        low : integer, float or string (i.e. extended trait name)
            The low end of the range.
        high : integer, float or string (i.e. extended trait name)
            The high end of the range.
        value : integer, float or string (i.e. extended trait name)
```

Advanced ETS: fancier UIs beyond TraitsUI

TraitsUI is limited: accessing the toolkit

TraitsUI is super simple to get started. It also abstracts away differences between wx and qt and their respective wrappers to provide that simple API. BUT:

- It does not offer pixel level controls over UIs. Other than regular spacing between items using Spring, there isn't almost anything.
- TraitsUI exposes a limited subset of the attributes/controls that Qt/Wx widgets have.

To go beyond TraitsUI capabilities, you can:

1. Access and modify the underlying Qt/Wx widget using a View's handler/controller.
2. Use `qt_binder` to mix and match TraitsUI Items and qt objects bound to Traits.

Both of these approaches provide access to the full scope of the underlying toolkit.

Accessing the toolkit with a handler

Building the view doesn't provide access to the underlying toolkit widgets. That's because all `Item`s' editors are really editor factories. The toolkit's widget (control) is created from the Editor using the factory pattern. But once the view is up, the actual widgets get created, and can be accessed, for example in the view's handler's `init()` method. For e.g., a Qt UI's style sheet can be used to control color, font, or padding, by adding a call to the `setStyleSheet` method of a QtWidget:

```
class HouseHandler(Handler):
    def init(self, info):
        qt_object = info.ui.control
        qt_object.setStyleSheet('background-color: green')

class House(HasTraits):
    address = Str
    bedrooms = Int
    view = View(Item('address', style="readonly"),
                Item('bedrooms'),
                handler=HouseHandler())
```

Advanced ETS: Mayavi

https://docs.enthought.com/mayavi/

Mayavi demos

Demos illustrates a wide array of tools and interactions to explore data (clone `mayavi` from github.com, go to `examples/` folder):

- `mayavi/advanced_visualization/mlab_3D_to_2D.py`
- `mayavi/advanced_visualization/polydata.py`

Resources:

- KBI Python users group on Microsoft Teams. Contact jrocher@kbibiopharma.com to join, ask question, read news, …
- Enthought Tool Suite documentation:
  - https://docs.enthought.com/ets/
  - https://docs.enthought.com/traits/
  - https://docs.enthought.com/traitsui/
  - https://docs.enthought.com/chaco/
  - https://qt-binder.readthedocs.io/en/latest/
  - ...
- Enthought Deployment Manager (EDM): https://www.enthought.com/product/enthought-deployment-manager/