From a Script to a Scalable Application with ETS

SciPy2022 tutorial 12 JUL 2022

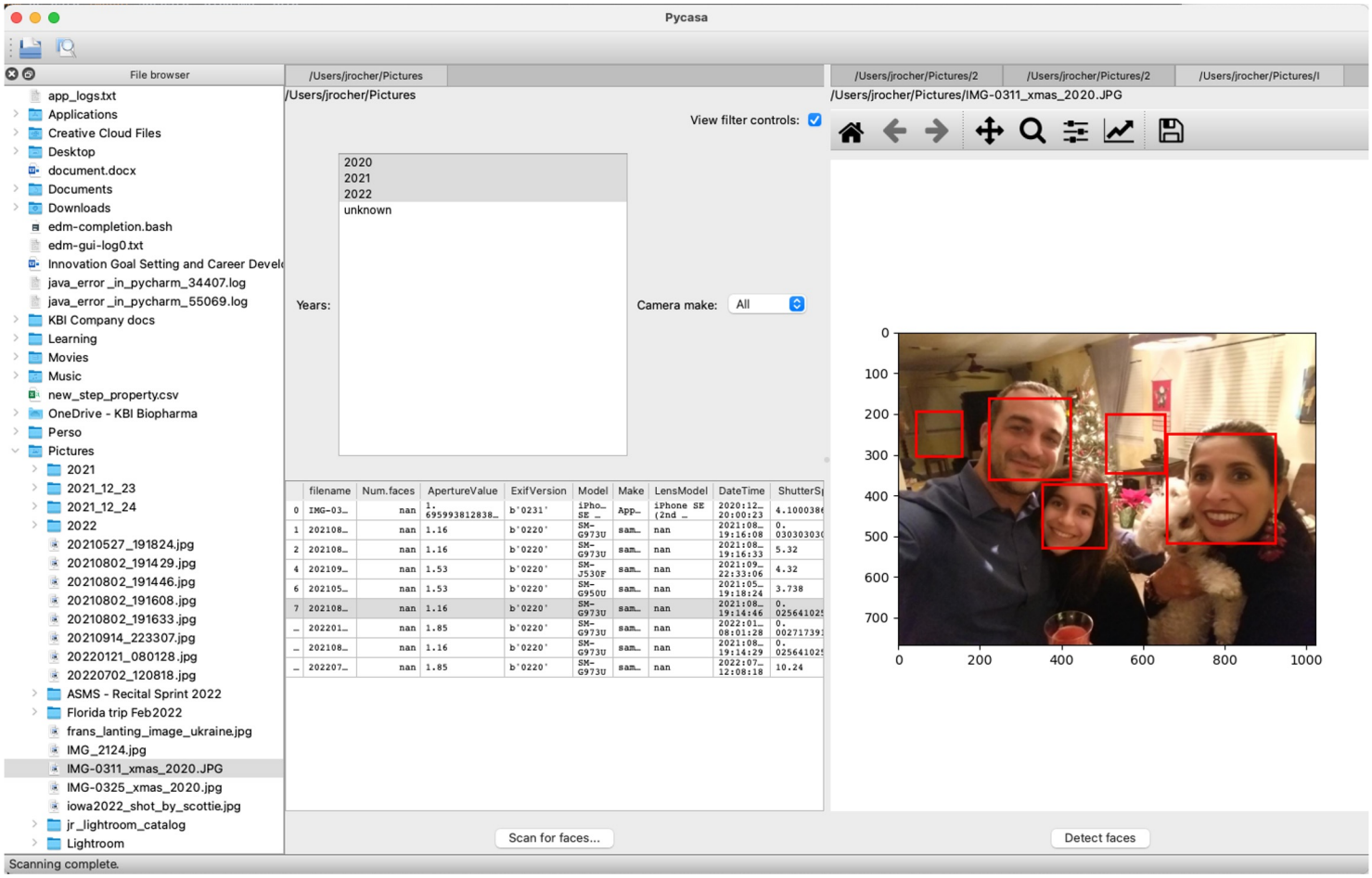
Jonathan Rocher[1], Prabhu Ramachandran[3,2], Siddhant Wahal[2], Jason Chambless[1], Corran Webster[2]

[1]KBI Biopharma Inc.
[2]Enthought Inc.
[3]ITT-Bombay

Outline

1. Foreword
2. The application stack: standing on the shoulders of giants
3. Introduction to ETS: Traits
4. Introduction to ETS: TraitsUI
5. Building a scalable application with Pyface
6. More resources

What we will build (3.5h)



https://github.com/jonathanrocher/ets_tutorial

What is 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 highly flexible 2D and 3D plotting frameworks 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/

Why/when ETS?

At KBI, here were our criteria for selecting ETS to build multiple scientific applications:

1. Limitless for data related tools (plotting, …)
2. Robust/mature
3. Development costs
4. Open-source, pro support if needed

What wasn't as important **to us** for **some** of our products:

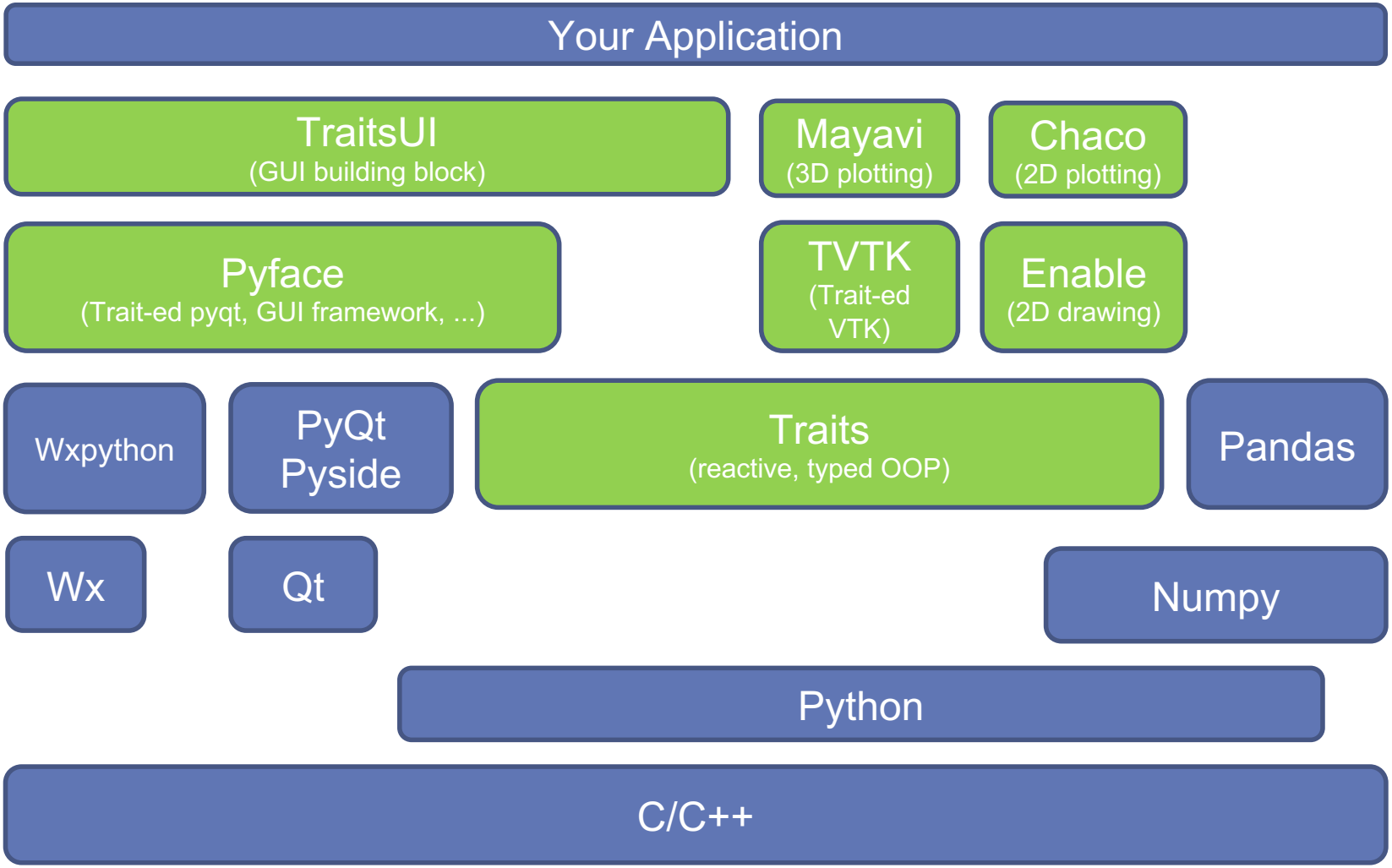
1. Desktop applications are hard to deploy and maintain with large number of users.
2. Qt-based means limited in GUI elements (no javascript-style swishy views)

Packages in ETS

1. traits
2. traitsUI
3. pyface
4. envisage
5. chaco, enable
6. Mayavi
7. scimath
8. traits_futures
9. qt_binder
10. apptools
11. block_context
12. …

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

ETS application stack



Intro to ETS: Traits

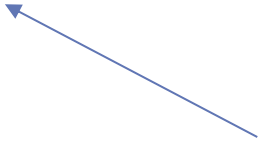
https://docs.enthought.com/traits/

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

```
import os

class FileRepository(object):
   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 (FileRepository) to create one of these “objects”, it is called an instance (repo):

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

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

Traits' listeners (the dynamic form)

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

class FileRepository(HasStricTraits):
    ...

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

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

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, HasStricTraits, Int,\
    List, observe, Property, Str

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

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

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

Trait initialization

Each trait of a Traits class has a default value based on its type, always evaluating to `False`. `Str` default to an empty string, `Int` to 0, `Dict`, `Set` and `List` to empty versions, ... Simple default values can be placed in declaration command. More complex defaults can be returned by a "_default" method.

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

class FileRepository(HasStrictTraits):
    name = Str
    url = Str(".")
    exp_list = List
    scanned = Bool
    num_exp = Int

    def _exp_list_default(self):
        if "tmp" in os.listdir(self.url):
            return ["tmp"]
        else:
            return []

>>> repo = FileRepository()
>>> repo.name

>>> repo.url
.
```

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.file_repository import FileRepository

class Analysis(HasStrictTraits):
    repository = Instance(FileRepository)
    size = Int

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

    @observe("repository")
    def when_repository_changed(self):
        ...

>>> repo = FileRepository()
>>> analysis = Analysis(repository=repo)
>>> analysis.summarize()
```

Working with `Instance` (*)

1. The default value is `None`, so don't forget to initialize.

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

Initialize with:

```
class Analysis(HasStrictTraits):
    ...
    def _repository_default(self):
        return FileRepository()
```

or equivalently

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

2. `Instance` can receive an importable path to a class rather than a class itself:

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

or

```
    repository = Instance("mypkg.model.file_repository.FileRepository")
```

3. The dynamic listeners are capable of listening to attributes of attributes:

```
class Analysis(HasStrictTraits):
    ...
    @observe("repository.scanned")
    def notify_when_repo_or_scanned_default(self, event):
        print("Either the repository has changed, or its scanned value.")

    @observe("repository:scanned")
    def notify_when_scanned_default(self, event):
        print("The scanned value of the repository attribute has changed.")
```

“Free” GUI for a traits model

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

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

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

    def export(self):
        ...

>>> repo = FileRepository(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 FileRepository(HasStricTraits):
    ...
    def traits_view(self):
        view = View(
            Item("name"),
            Item("url")
            Item("exp_list"),
        )
        return view
```

Add actions to the GUI

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

class FileRepository(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()
```

Controlling layout with TraitsUI

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

class FileRepository(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 FileRepository(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.

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 FileRepository(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
```

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
   `visible_when`: allow dynamic views with groups of widgets appearing based on boolean expressions.
3. `View` class:
   `width`, `height`, `resizable`: control the size of the window
   `title`, `icon`: title of the window
   `scrollable`: whether to force the container to expand or make the panel scrollable.
   `buttons`: list of buttons for the master view (OK, Cancel, customs, ...)
   `handler`, `key_bindings`: control the behavior of the view

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, HasStricTraits, \
    Int, List, Str

class MFIFileRepository(HasStricTraits):
    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()
```

Controlling window buttons

```
from traits.api import Bool, HasStrictTraits
from traitsui.api import CancelButton, Item, OKButton, OKCancelButtons,
View

class Test(HasStrictTraits):
    attr = Bool
    view = View(
        Item("attr"),
        buttons=OKCancelButtons
    )
```

or equivalently

```
class Test2(HasStrictTraits):
    attr = Bool
    view = View(
        Item("attr"),
        buttons=[OKButton, CancelButton]
    )
```

By default, both of these buttons, both button implementations close the dNote: See the controller section for customizing what happens, or creating custom buttons.

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.

Advanced TraitsUI

The Controller

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

class FileRepositoryView(ModelView):
   model = Instance(FileRepository)
   scan_button = Event

   def traits_view(self):
      view = View(
         Item("model.name"),
         Item("model.url", label="Repo URL"),
         Item("model.exp_list", style="readonly"),
         title= "Repository manager",
         resizable=True
         buttons=[ScanButton, OKButton, CancelButton]
         handler=FileRepositoryHandler()
      )
      return view

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

```
>>> model = FileRepository(url="path")
>>> view = FileRepositoryView(model=model)
>>> view.configure_traits()
```

The Controller

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


ScanButton = Action(name='Scan', action="do_scan",
                    enabled_when="can_scan")


class FileRepositoryHandler(Handler):
   def do_scan(self, info):
      info.object.scan_button = True
```

Nesting views with `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 App(HasStrictTraits)
    repository = Instance(FileRepositoryView)
    ...
    view = View(
         Item("repository", editor=InstanceEditor(), style="custom"),
         ...
         title="MFI App"
    )


class FileRepositoryView(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

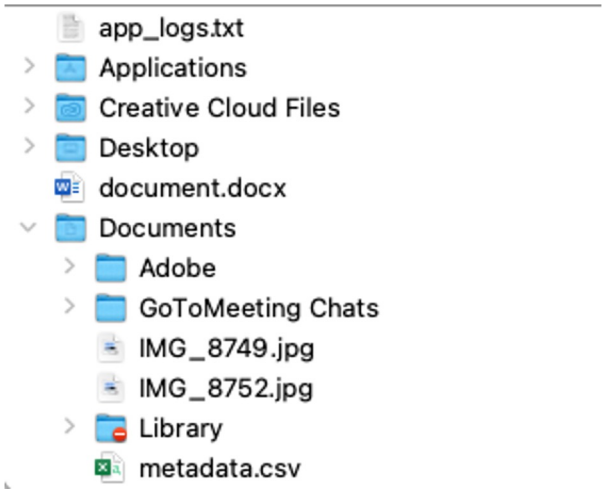
```
from traits.api import Array, Instance
from traitsui.api import ArrayEditor, Item, View
from traitsui.ui_editors.data_frame_editor import DataFrameEditor

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"),
        )
```

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.
- `DateEditor`, `TimeEditor`
- `MPLFigureEditor`
- …

For more examples and demos, see https://docs.enthought.com/traitsui/demos .

Menus and toolbars

It can be done in pure TraitsUI though I recommend to use the pyface framework if you need to build a real application with menus and toolbar. See pyface section

TraitsUI's Controller: custom button

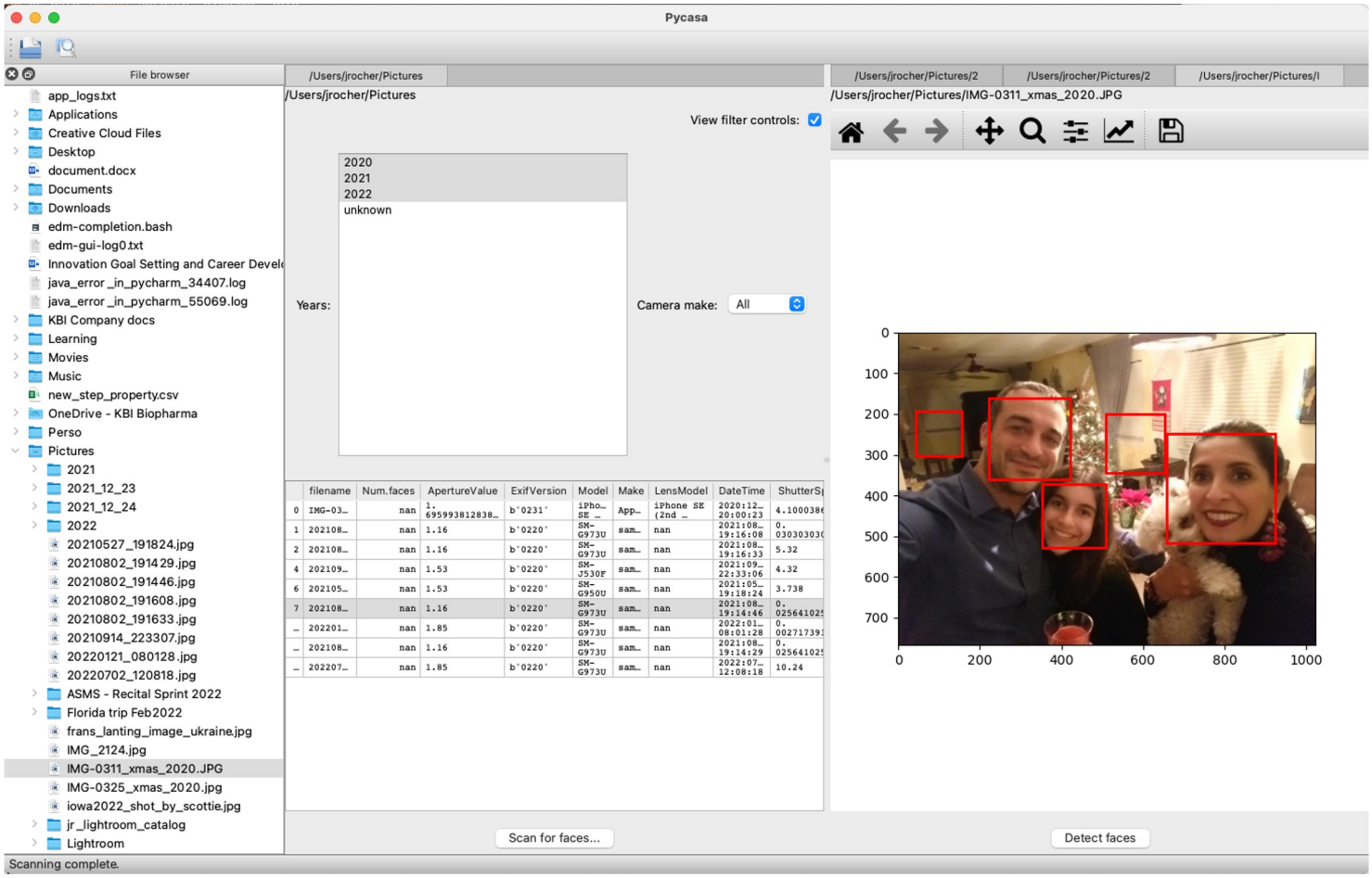
```
from traits.api import Button, Instance
from traitsui.api import Action, Handler, Item, ModelView, OKButton, View

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):
        ...
```

What we are building

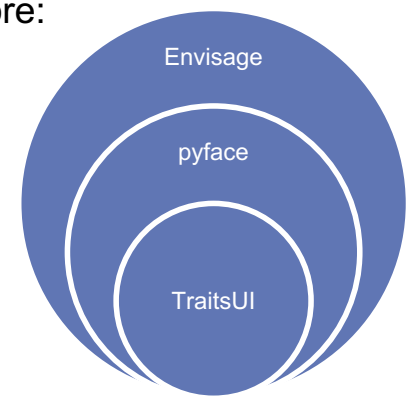


ETS's application frameworks

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:



Advanced ETS: pyface

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

What is pyface?

- `pyface` is a key package in ETS. It can globally be thought of 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` isn't well documented and certain parts are deprecated (`workbench` for example). Therefore this tutorial!
- Valuable components of pyface for app development:
  1. Quick native dialogs (**`error, warning, information, confirm, ...`**)
  2. The "tasks" framework for mid-size application building. Can be coupled with Envisage for large-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:

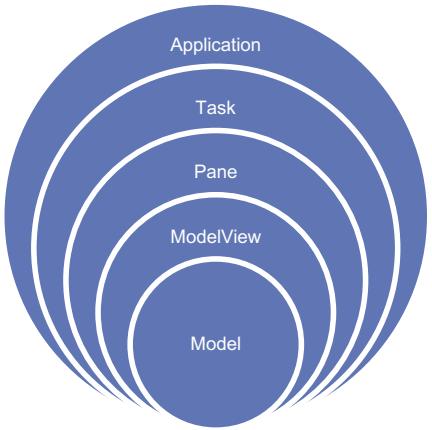
```
from pyface.api import FileDialog, OK
file_dialog = FileDialog(title='Export User Data',
                         action='save as',
                         wildcard="My Data (*.txt)|*.txt")
file_dialog.open()
if file_dialog.return_code == OK:
    path = file_dialog.path
    ...
```

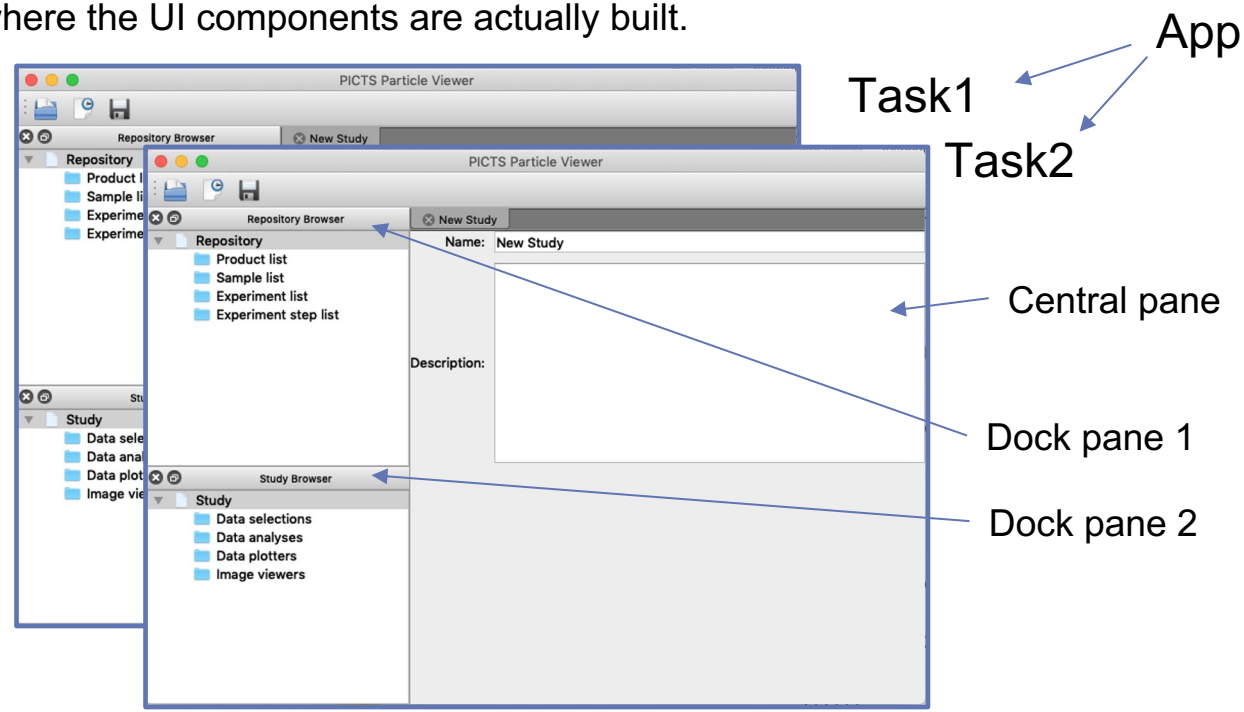
https://docs.enthought.com/pyface/

`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 window, or shown/hidden, made of TraitsUI views (`pyface.tasks.task_pane.TaskPane`).
4. UI components (traitsUI): where the UI components are actually built.





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()
```

Interlude: GUI Application project structure

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

In each sub-package:

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

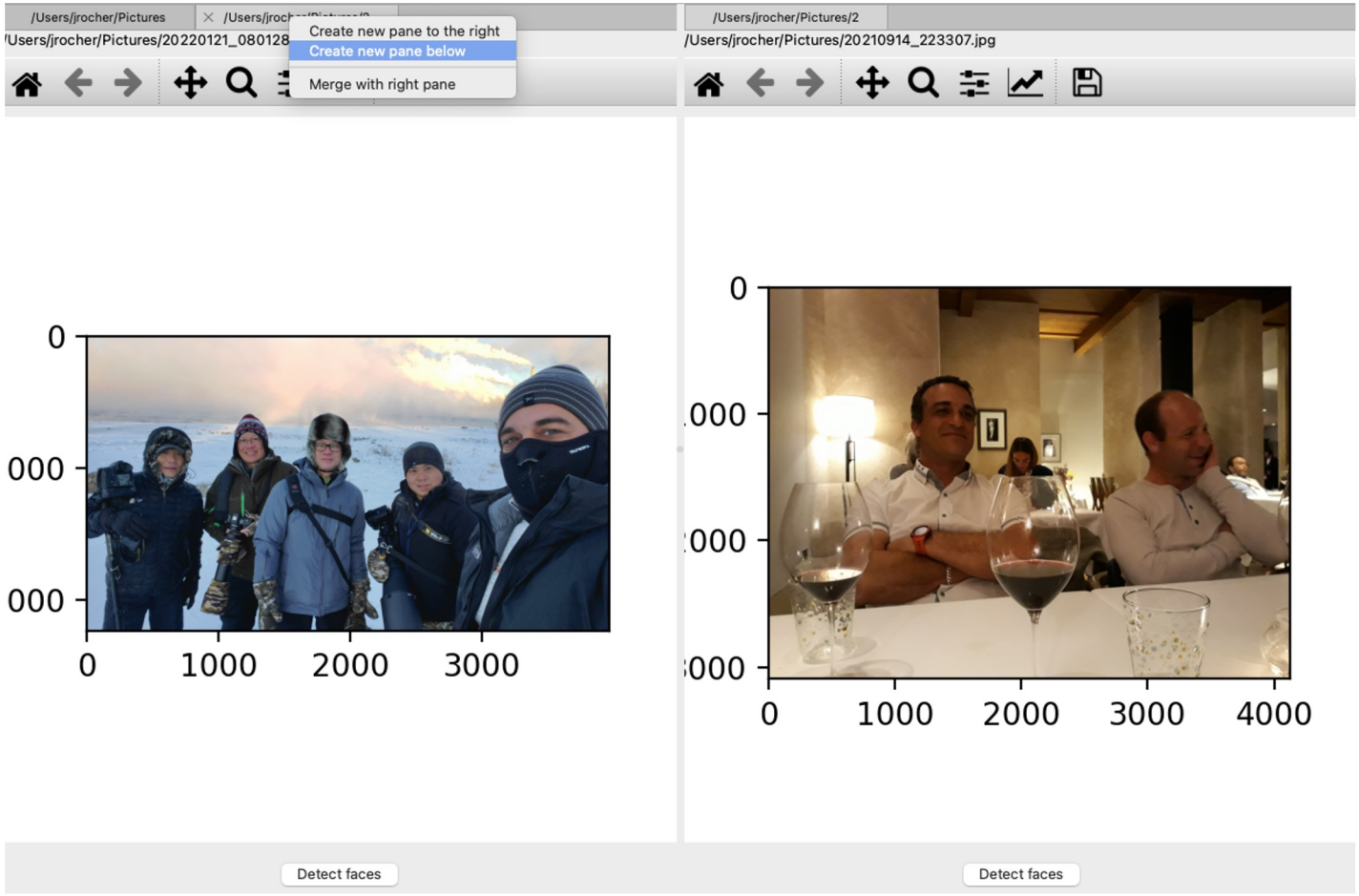
pyface's Task app: multiple panes

Applications of any reasonable size will have more than 1 pane. So you will want to also create dock panes which will be laid out around it.

```
from pyface.tasks.api import Task, TraitsDockPane, TraitsTaskPane


class SidePane(TraitsDockPane):
    id = 'side_pane'
    def traits_view(self):
        return View(
            Item(...)
        )


class HelloWorldTask(Task):
    def create_central_pane(self):
        return HelloWorldPane()

    def create_dock_panes(self)
        return [SidePane()]

    def _default_layout_default(self):
        return TaskLayout(
            left=PaneItem('side_pane', width=300)
```

A good central pane

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



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
```

Beyond “hello world”: important attributes and methods

For the tasks and panes, the following attributes can be overloaded or customized:

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)
- `default_layout`

Pane’s interface:

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

SplitEditorAreaPane:

- `active_editor`

Exercise: from stage4 to stage 5

Let's build the first real application for pycasa by doing the following:

1. Transform the Hello World example (stage 4.0) into something that's about pycasa. Use a SplitEditorAreaPane as the central pane. Create a method called `open_in_central_pane` in the task which can receive a filepath and open it in the central pane using the ImageFileView from stage 3.
    a. Reminder: to display a traits view as a tab, you need an Editor.
2. Add a `TraitsDockPane` with a file browser. To build this, create a FileBrowser model that holds a root path (`Directory`), and a `ModelView` to show it.
    a. Tip: a Directory, like a File can be displayed using the `FileEditor` in style "`custom`".
3. Add a listener on the double-click event of the view's file editor so that the task can open that file path in the central pane.

Bonus questions:

1. Add a button to the ImageFileView so that users can trigger the face recognition from within the central pane.
2. Add the ability to double click on folders too, and for that to display all the metadata of the images inside it. Use the DataFrameEditor to display that metadata.

Beyond “hello world”: important Application attributes and 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.

Life-cycle attributes and methods:

- `start()`
- `create_task(), create_window()`
- `close()`
- `tasks, windows`
- `task_factories`
- `active_task`

`Branding attributes:`

- `icon`
- `splash_screen`
- `extra_actions`

A note on ImageResource

Task toolbar entries, Application icons, Application splash screens and many other image related elements in pyface all expect a pyface ImageResource. For example,

In app.py

```
from pyface.api import ImageResource

class MyApp(Application):
    ...

    def _icon_default(self):
        return ImageResource('app_logo.png')
```

Supporting folder structure:

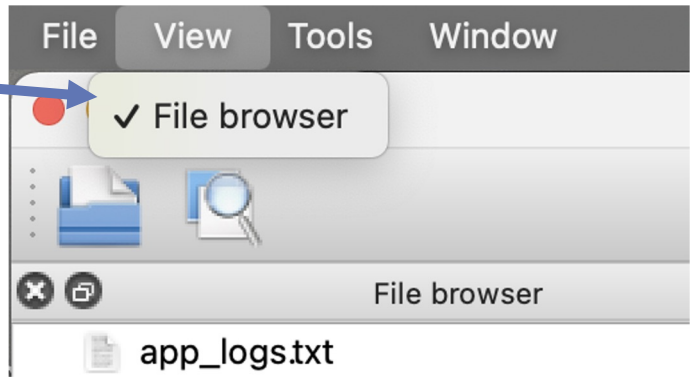
```
.../app.py
.../images/app_logo.png
```

Adding menus and toolbars

The `Task`'s `menu_bar` and `tool_bars` attributes can be set to create menus and toolbars. (Note that there is only 1 menu bar, but any number of tool bars, there the "s".)

```
from pyface.tasks.action.api import DockPaneToggleGroup, SGroup, SMenu, \
    SMenuBar, SToolBar, TaskAction, TaskWindowAction

class MyTask(Task):
   ...
   def _tool_bars_default(self):
      tool_bars = [
         SToolBar(
             TaskAction(name='Open...',
                        accelerator='Ctrl+N',
                        method='request_open_new_path',
                        image=ImageResource('document-open')),
         )
      ]
      return tool_bars

   def request_open_new_path(self):
      ...
```

```
...
def _menu_bar_default (self):
    menu_bar = SMenuBar(
     SMenu(
        SGroup(
            TaskAction(name='Open...',
                       accelerator='Ctrl+N',
                       method='request_open_new_path'),
            id='OpenGroup', name='OpenGroup',
        ),
        SGroup(
            TaskWindowAction(
                name='Close',
                accelerator='Ctrl+W',
                method='close'),
            id='CloseGroup', name='CloseGroup',
        ),
        id='File', name='&File'),
      SMenu(DockPaneToggleGroup(),
          id='View', name='&View'),
      ...
    )
    return menu_bar
```



Exercise: from stage 5 to stage 6

Starting from your output of the previous exercise or from stage 5.3, add the following:

1. A File > Open… menu entry and toolbar entry which prompts for a file/folder path and opens it in the central pane.
2. A Tools > Scan menu entry and toolbar entry which triggers a scan for faces for the path viewed in the currently active tab of the central pane.
3. A splash screen image and an icon for the application.

Bonus:

1. Add a checkbox to the path selector so if checked, faces are automatically scanned for after opening in the central pane.
2. Add a window status bar to the tasks which notifies users when face scanning is completed.

Adding app level menus

Actions requiring Application involvement, for example to control the creation or destruction of tasks/windows, must be injected into tasks menus by the Application through its `extra_actions` attribute. Additional actions can be injected into a menu, even into a group, by the application at task creation.

```
from pyface.action.schema.api import SchemaAddition

class MyApp(Application):
    ...
    def _extra_actions_default(self):
        actions = [
            SchemaAddition(id='custom_menu',
                factory=self.custom_method,
                path="MenuBar/File/GroupName",
                absolute_position="first")
        ]
        return actions

    def custom_method(self):
        <custom stuff only the app can/should do>
```

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`:

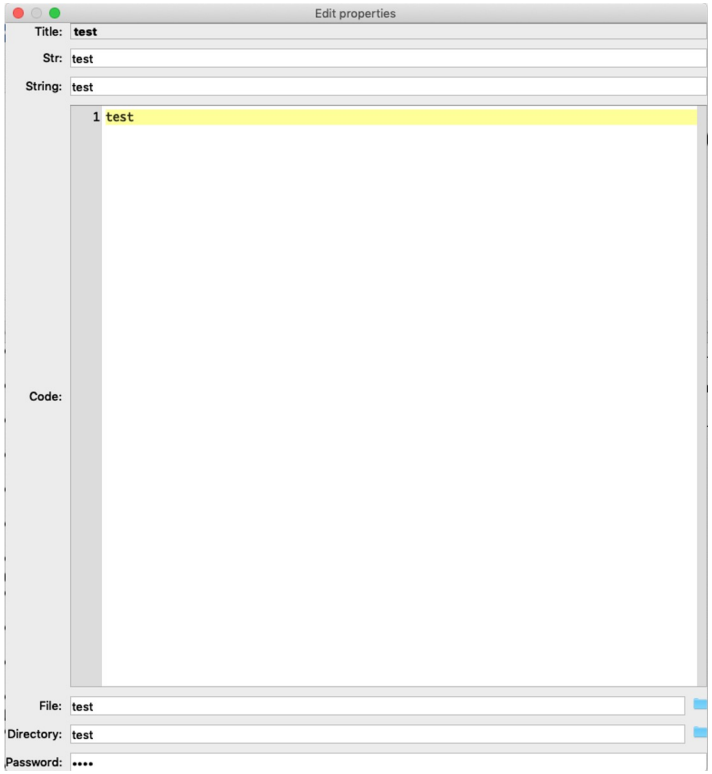
```
In [1]: from traits.trait_types import Code
```

```
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:

```
2152
2153    class Range(BaseRange):
2154        """ Defines a trait whose numeric value must be in a specified range using
2155            a C-level fast validator.
2156        """
2157
2158        def init_fast_validator(self, *args):
2159            """ Set up the C-level fast validator.
2160            """
2161            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:

```
1801    class BaseRange(TraitType):
1802        """ Defines a trait whose numeric value must be in a specified range.
1803        """
1804
1805        def __init__(
1806            self,
1807            low=None,
1808            high=None,
1809            value=None,
1810            exclude_low=False,
1811            exclude_high=False,
1812            **metadata
1813        ):
1814            """ Creates a Range trait.
1815
1816            Parameters
1817            ----------
1818            low : integer, float or string (i.e. extended trait name)
1819                The low end of the range.
1820            high : integer, float or string (i.e. extended trait name)
1821                The high end of the range.
1822            value : integer, float or string (i.e. extended trait name)
```

Optional TraitUI editor attributes

Another way to customize behavior further is to set optional attributes on the TraitUI editor used. For example, when using a `DataFrameEditor`, jumping (using a “smart” code editor) to the definition of that class provides quick access to the list of attributes that can be set:

Application packaging and sharing

Options for distributing Applications

Coder friendly:

1. `git + edm/conda/pip` for sharing an application with other developers.
2. `setup.py` for generating an egg/wheel and defining entry points (for users comfortable with command line). Done in stage 8.1.

Commercial

1. `edm, conda` have great commercial options.

Open-source 1-click installers

1. `wixtoolset` (https://wixtoolset.org/) full-featured, open-source but Windows only.
2. `pyinstaller` (https://pyinstaller.org) open source, cross-platform
3. `cx_freeze` open source, cross-platform (https://cx-freeze.readthedocs.io/)

…

Distributing an app: PyInstaller

PyInstaller packages a script (or set of files) into a distributable application

To install with pip:

```
pip install -U pyinstaller
```

Then change to the directory of your program and run one of these commands:

`pysinstaller --windowed my_script.py` (for windowed applications)

`pysinstaller --console my_script.py` (for console applications)

Two output types:

- One-Folder (default): folder with all dependencies and an executable
- One File: only single executable file (`--onefile` option)

The resulting bundle or file can be distributed without the need to install any modules or particular versions of Python

Distributing an app: PyInstaller

For a detailed description of using PyInstaller, see the ReadMe in
`stage8.2 packaging pyinstaller`

More advanced applications with complex dependencies will require additional options:

- Using Spec Files to bundle data, include run-time libraries, add run-time options
- Using Hook Files to help PyInstaller locate hard-to-find or unusually imported dependencies

Extensive documentation exists at https://pyinstaller.org, including what to do when things go wrong:

- When Things Go Wrong

5. More Resources

- Enthought Tool Suite documentation:
  - https://docs.enthought.com/ets/
  - https://docs.enthought.com/traits/
  - https://docs.enthought.com/traitsui/
  - https://qt-binder.readthedocs.io/
  - ...
- ETS mailing list: `ets-users` on google groups
- Enthought Deployment Manager (EDM): https://www.enthought.com/product/enthought-deployment-manager/