Orange Development Documentation

Release 3

Orange Data Mining

Sep 19, 2018

Contents

CHAPTER 1

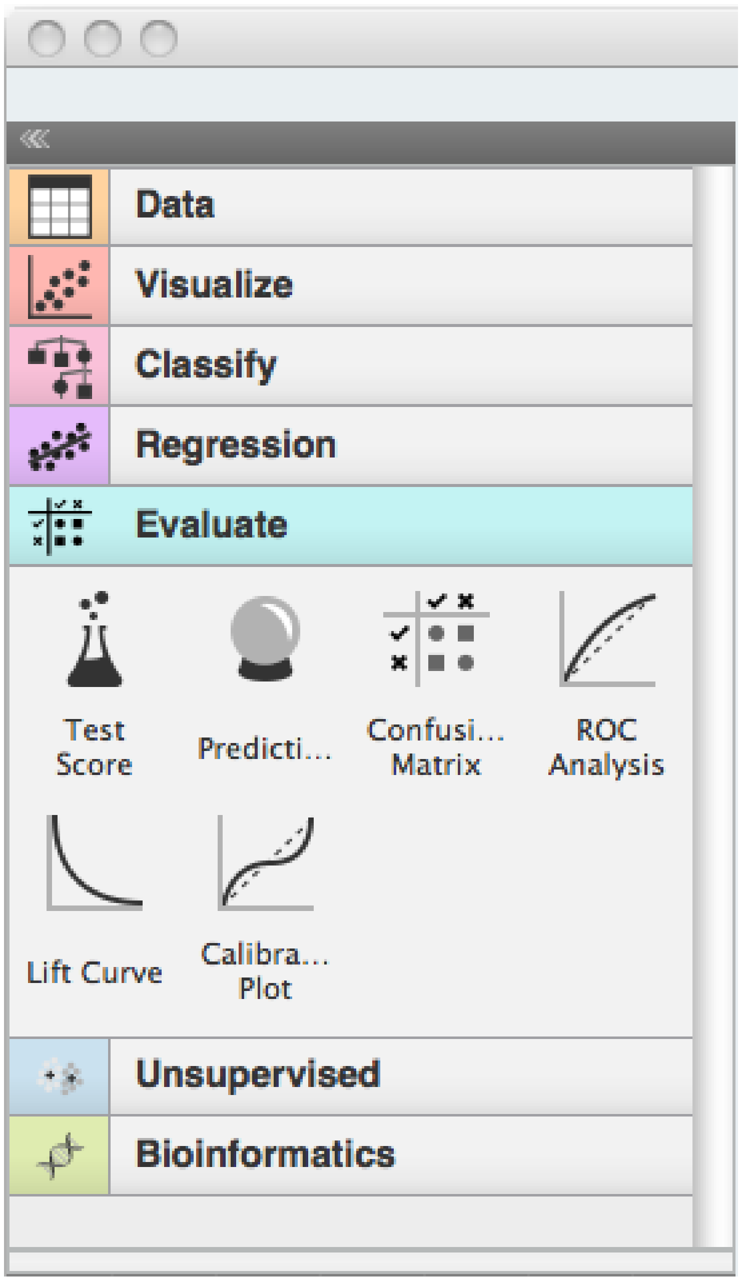
Getting Started

Orange Widgets are components in Orange Canvas, a visual programming environment of Orange. They represent some self contained functionalities and provide a graphical user interface (GUI). Widgets communicate with each other and pass objects through communication channels to interact with other widgets.

On this page, we will start with some simple essentials, and then show you how to build a simple widget that will be ready to run within Orange Canvas.

1.1 Prerequisites

Each Orange widget belongs to a category and has an associated priority within that category. When opening Orange Canvas, a visual programming environment that comes with Orange, widgets are listed in a toolbox on the left:



Each widget has a name description and a set of input/outputs (referred to as the widget's meta description).

This meta data is discovered at Orange Canvas application startup leveraging setuptools/distribute and its entry points protocol. Orange Canvas looks for widgets using an `orange.widgets` entry point.

1.2 Defining a widget

`OWWidget` is the base class of a widget in the Orange Canvas workflow.

Every widget in the canvas framework needs to define its meta data. This includes the widget's name and text descriptions and more importantly its input/output specification. This is done by defining constants in the widget's class namespace.

We will start with a very simple example. A widget that will output a single integer specified by the user.

```
from Orange.widgets.widget import OWWidget, Output
from Orange.widgets import gui

class IntNumber(OWWidget):
    # Widget's name as displayed in the canvas
    name = "Integer Number"
    # Short widget description
    description = "Lets the user input a number"

    # An icon resource file path for this widget
    # (a path relative to the module where this widget is defined)
    icon = "icons/number.svg"

    # Widget's outputs; here, a single output named "Number", of type int
    class Outputs:
        number = Output("Number", int)
```

By design principle, Orange widgets in an interface are most often split to control and main area. Control area appears on the left and should include any controls for settings or options that your widget will use. Main area would most often include a graph, table or some drawing that will be based on the inputs to the widget and current options/setting in the control area. `OWWidget` makes these two areas available through its attributes `self.controlArea` and `self.mainArea`. Notice that while it would be nice for all widgets to have this common visual look, you can use these areas in any way you want, even disregarding one and composing your widget completely unlike the others in Orange.

We specify the default layout with class attribute flags. Here we will only be using a single column (controlArea) GUI.

```
# Basic (convenience) GUI definition:
#   a simple 'single column' GUI layout
want_main_area = False
#   with a fixed non resizable geometry.
resizing_enabled = False
```

We want the current number entered by the user to be saved and restored when saving/loading a workflow. We can achieve this by declaring a special property/member in the widget's class definition like so:

```
number = Setting(42)
```

And finally the actual code to define the GUI and the associated widget functionality:

```
def __init__(self)
    super().__init__()

    gui.lineEdit(self.controlArea, self, "number", "Enter a number",
                 box="Number",
                 callback=self.number_changed,
                 valueType=int, validator=QIntValidator())
    self.number_changed()

def number_changed(self):
    # Send the entered number on "Number" output
    self.Outputs.number.send(self.number)
```

See also:

`Orange.widgets.gui.lineEdit()`,

By itself this widget is useless because no widget accepts its output. So let us define a widget that displays a number.

```python
from Orange.widgets.widget import OWWidget, Input
from Orange.widgets import gui

class Print(widget.OWWidget):
    name = "Print"
    description = "Print out a number"
    icon = "icons/print.svg"

    class Inputs:
        number = Input("Number", int)

    want_main_area = False

    def __init__(self):
        super().__init__()
        self.number = None

        self.label = gui.widgetLabel(self.controlArea, "The number is: ??")

    @Inputs.number
    def set_number(self, number):
        """Set the input number."""
        self.number = number
        if self.number is None:
            self.label.setText("The number is: ??")
        else:
            self.label.setText("The number is {}".format(self.number))
```

We define inputs with a class *Inputs*, just like outputs are defined by *Outputs*. However, each input must be handled by a class methods. We mark the handlers by decorating them; in above case by putting *@Inputs.number* before the method's definition.

Notice how in the *set_number* method we check whether the number is *None*. *None* is sent to the widget when a connection between the widgets is removed or if the sending widget to which we are connected intentionally emptied the channel.

Now we can use one widget to input a number and another to display it.

One more:

```python
from Orange.widgets.widget import OWWidget, Input, Output

class Adder(OWWidget):
    name = "Add two integers"
    description = "Add two numbers"
    icon = "icons/add.svg"

    class Inputs:
        a = Input("A", int)
        b = Input("B", int)

    class Outputs:
        sum = Output("A + B", int)

    want_main_area = False

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        super().__init__()
        self.a = None
        self.b = None

    @Inputs.a
    def set_A(self, a)
        """Set input 'A'."""
        self.a = a

    @Inputs.b
    def set_B(self, b):
        """Set input 'B'."""
        self.b = b

    def handleNewSignals(self):
        """Reimplemeted from OWWidget."""
        if self.a is not None and self.b is not None:
            self.Outputs.sum.send(self.a + self.b)
        else:
            # Clear the channel by sending `None`
            self.Outputs.sum.send(None)
```

See also:

`handleNewSignals()`

CHAPTER 2

Getting Started (Continued)

After learning what an Orange Widget is and how to define them on a toy example, we will build an semi-useful widgets that can work together with the existing Orange Widgets.

We will start with a very simple one, that will receive a dataset on the input and will output a dataset with 10% of the data instances. We will call this widget *OWDataSamplerA* (OW for Orange Widget, DataSampler since this is what widget will be doing, and A since we prototype a number of this widgets in our tutorial).

2.1 A 'Demo' package

First in order to include our new widgets in the Orange Canvas's toolbox we will create a dummy python project named *orange-demo*

The layout should be:

```
orange-demo/
      setup.py
      orangedemo/
                  __init__.py
                  OWDataSamplerA.py
```

and the `orange-demo/setup.py` should contain

```
from setuptools import setup

setup(name="Demo",
      packages=["orangedemo"],
      package_data={"orangedemo": ["icons/*.svg"]},
      classifiers=["Example :: Invalid"],
      # Declare orangedemo package to contain widgets for the "Demo" category
      entry_points={"orange.widgets": "Demo = orangedemo"},
      )
```

Note that we declare our *orangedemo* package as containing widgets from an ad hoc defined category *Demo*.

See also:

https://github.com/biolab/orange3/wiki/Add-Ons

Following the previous examples, our module defining the OWDataSamplerA widget starts out as:

```
import sys
import numpy

import Orange.data
from Orange.widgets.widget import OWWidget, Input, Output
from Orange.widgets import gui

class OWDataSamplerA(OWWidget):
    name = "Data Sampler"
    description = "Randomly selects a subset of instances from the dataset"
    icon = "icons/DataSamplerA.svg"
    priority = 10

    class Inputs:
        data = Input("Data", Orange.data.Table)

    class Outputs:
        sample = Output("Sampled Data", Orange.data.Table)

    want_main_area = False

    def __init__(self):
        super().__init__()

        # GUI
        box = gui.widgetBox(self.controlArea, "Info")
        self.infoa = gui.widgetLabel(box, 'No data on input yet, waiting to get␣
↪something.')
        self.infob = gui.widgetLabel(box, '')
```

The widget defines an input channel “Data” and an output channel called “Sampled Data”. Both will carry tokens of the type `Orange.data.Table`. In the code, we will refer to the signals as *Inputs.data* and *Outputs.sample*.

Channels can carry tokens of arbitrary types. However, the purpose of widgets is to talk with other widgets, so as one of the main design principles we try to maximize the flexibility of widgets by minimizing the number of different channel types. Do not invent new signal types before checking whether you cannot reuse the existing.

As our widget won’t display anything apart from some info, we will place the two labels in the control area and surround it with the box “Info”.

The next four lines specify the GUI of our widget. This will be simple, and will include only two lines of text of which, if nothing will happen, the first line will report on “no data yet”, and second line will be empty.

In order to complete our widget, we now need to define a method that will handle the input data. We will call it `set_data()`; the name is arbitrary, but calling the method *set_<the name of the input>* seems like a good practice. To designate it as the method that accepts the signal defined in *Inputs.data*, we decorate it with *@Inputs.data*.

```
    @Inputs.data
    def set_data(self, dataset):
        if dataset is not None:
            self.infoa.setText('%d instances in input dataset' % len(dataset))
            indices = numpy.random.permutation(len(dataset))
            indices = indices[:int(numpy.ceil(len(dataset) * 0.1))]
```

(continues on next page)

(continued from previous page)

```
            sample = dataset[indices]
            self.infob.setText('%d sampled instances' % len(sample))
            self.Outputs.sample.send(sample)
        else:
            self.infoa.setText('No data on input yet, waiting to get something.')
            self.infob.setText('')
            self.Outputs.sample.send("Sampled Data")
```

The `dataset` argument is the token sent through the input channel which our method needs to handle.

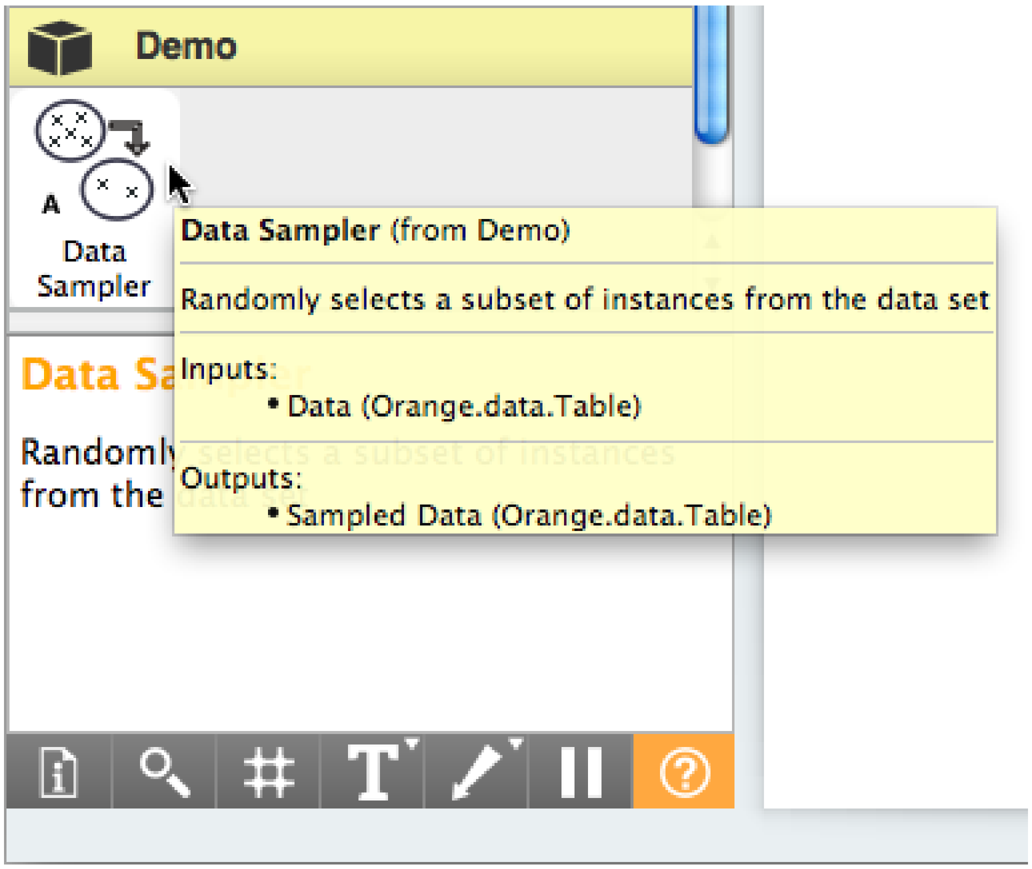
To handle a non-empty token, the widget updates the interface reporting on number of data items on the input, then does the data sampling using Orange's routines for these, and updates the interface reporting on the number of sampled instances. Finally, the sampled data is sent as a token to the output channel defined as *Output.sample*.

Although our widget is now ready to test, for a final touch, let's design an icon for our widget. As specified in the widget header, we will call it `DataSamplerA.svg` and put it in *icons* subdirectory of *orangedemo* directory.

With this we can now go ahead and install the orangedemo package. We will do this by running `pip install -e .` command from within the *orange-demo* directory.

Note: Depending on your python installation you might need administrator/superuser privileges.

For a test, we now open Orange Canvas. There should be a new pane in a widget toolbox called Demo. If we click on this pane, it displays an icon of our widget. Try to hover on it to see if the header and channel info was processed correctly:



Data Sampler (from Demo)

Randomly selects a subset of instances from the data set

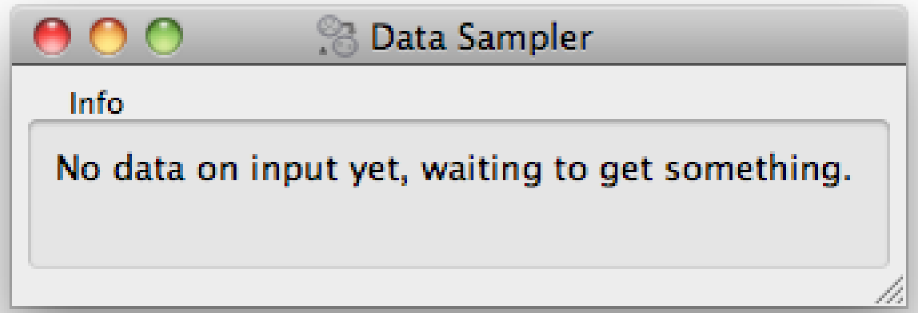
Inputs:

- Data (Orange.data.Table)

Outputs:

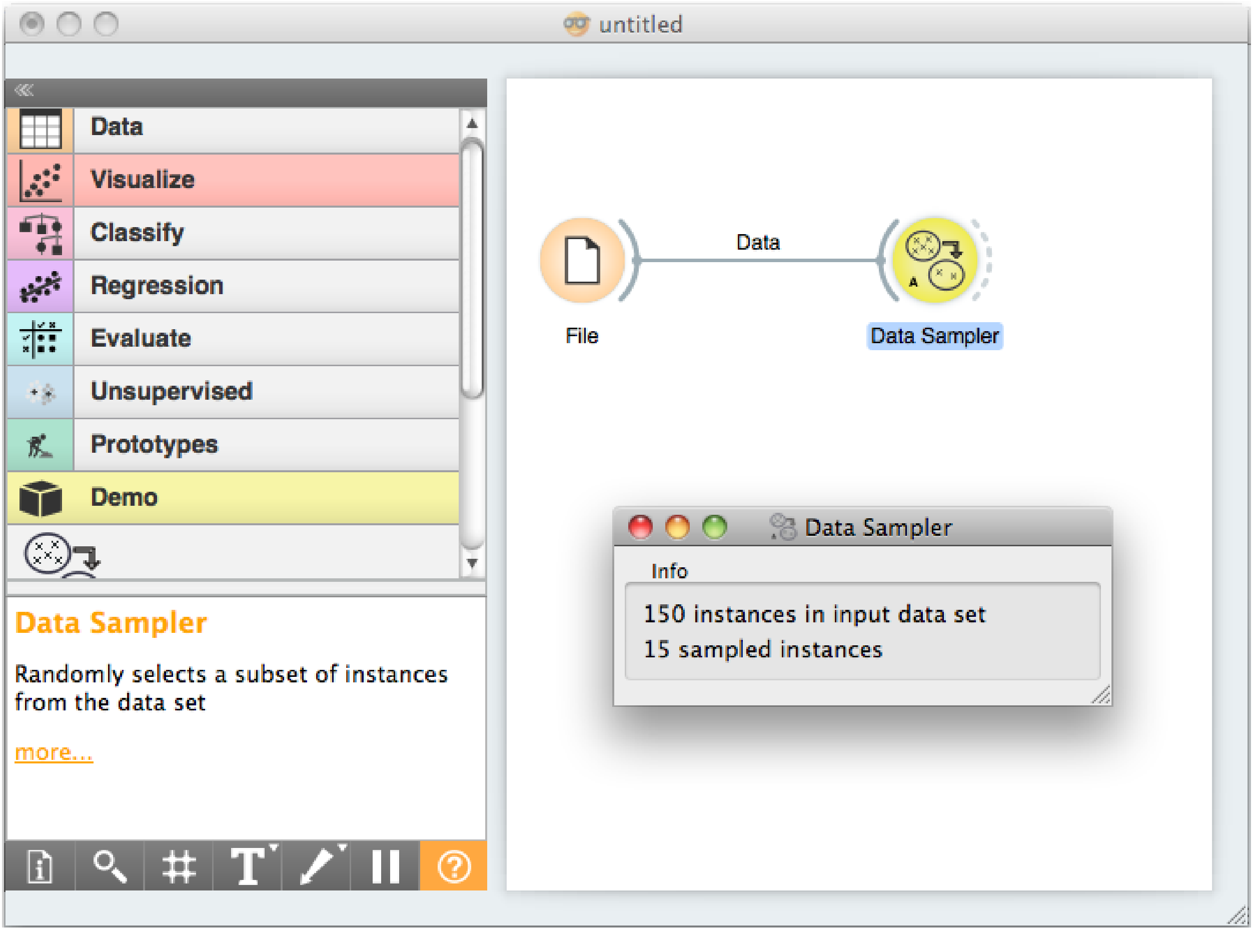
- Sampled Data (Orange.data.Table)

Now for the real test. We put the File widget on the schema (from Data pane) and load the iris.tab dataset. We also put our Data Sampler widget on the scheme and open it (double click on the icon, or right-click and choose Open):



No data on input yet, waiting to get something.

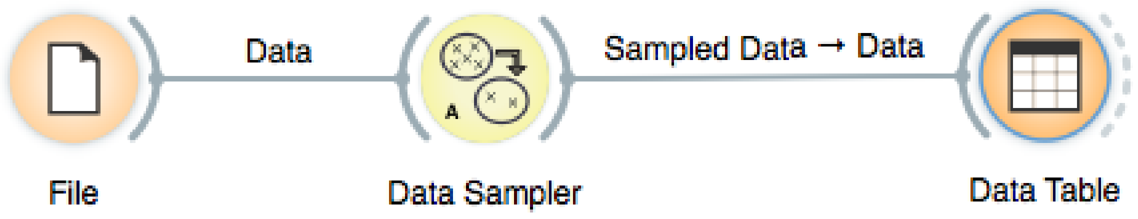
Now connect the File and Data Sampler widget (click on an output connector of the File widget, and drag the line to the input connector of the Data Sampler). If everything is ok, as soon as you release the mouse, the connection is established and, the token that was waiting on the output of the file widget was sent to the Data Sampler widget, which in turn updated its window:



Randomly selects a subset of instances from the data set

150 instances in input data set
15 sampled instances

To see if the Data Sampler indeed sent some data to the output, connect it to the Data Table widget:



Try opening different data files (the change should propagate through your widgets and with Data Table window open, you should immediately see the result of sampling). Try also removing the connection between File and Data Sampler (right click on the connection, choose Remove). What happens to the data displayed in the Data Table?

2.1.1 Testing Your Widget Outside Orange Canvas

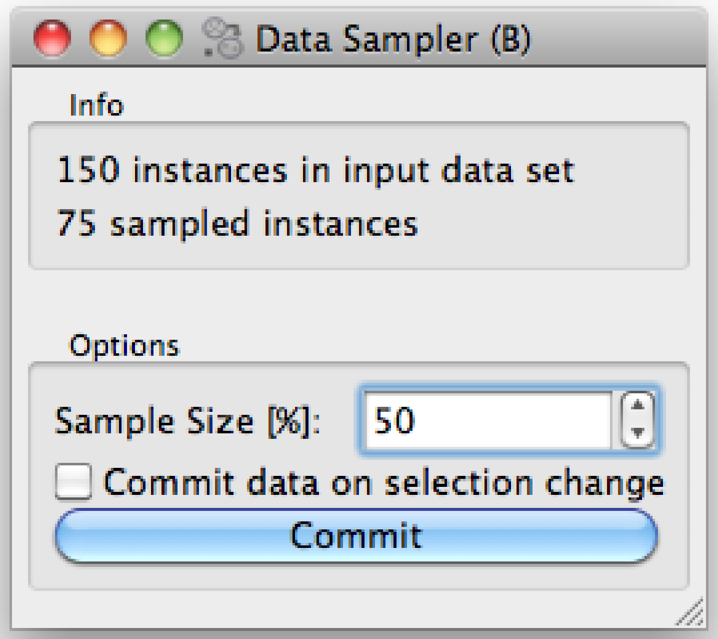
As a general rule each widget should have a simple *main* stub function so it can be run independently from Orange Canvas

```
def main(argv=None):
    from AnyQt.QtWidgets import QApplication
    # PyQt changes argv list in-place
    app = QApplication(list(argv) if argv else [])
    argv = app.arguments()
    if len(argv) > 1:
        filename = argv[1]
    else:
        filename = "iris"

    ow = OWDataSamplerA()
    ow.show()
    ow.raise_()

    dataset = Orange.data.Table(filename)
    ow.set_data(dataset)
    ow.handleNewSignals()
    app.exec_()
    ow.set_data(None)
    ow.handleNewSignals()
    ow.onDeleteWidget()
    return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

CHAPTER 3

Tutorial (Settings and Controls)

In the *previous section* of our tutorial we have just built a simple sampling widget. Let us now make this widget a bit more useful, by allowing a user to set the proportion of data instances to be retained in the sample. Say we want to design a widget that looks something like this:



150 instances in input data set
75 sampled instances

Sample Size [%]: 50
Commit data on selection change

What we added is an Options box, with a spin entry box to set the sample size, and a check box and button to commit (send out) any change we made in setting. If the check box with "Commit data on selection change" is checked, than any change in the sample size will make the widget send out the sampled dataset. If datasets are large (say of several thousands or more) instances, we may want to send out the sample data only after we are done setting the sample size, hence we left the commit check box unchecked and press "Commit" when we are ready for it.

This is a very simple interface, but there is something more to it. We want the settings (the sample size and the state of the commit button) to be saved. That is, any change we made, we want to save it so that the next time we open the widget the settings is there as we have left it

3.1 Widgets Settings

Luckily, since we use the base class `OWWidget`, the settings will be handled just fine. We only need to tell which variables we want to use for persistent settings.

In our widget, we will use two settings variables, and we declare this in the widget class definition (after the *inputs*, *outputs* definitions).

```
class OWDataSamplerB(OWWidget):
    name = "Data Sampler (B)"
    description = "Randomly selects a subset of instances from the dataset."
    icon = "icons/DataSamplerB.svg"
    priority = 20

    class Inputs:
        data = Input("Data", Orange.data.Table)

    class Outputs:
        sample = Output("Sampled Data", Orange.data.Table)

    proportion = settings.Setting(50)
    commitOnChange = settings.Setting(0)
```

All settings have to specify their default value. When a widget is created the widget's members are already restored and ready to use in its *__init__* method. The contents of the two variables (`self.proportion` and `self.commitOnChange`) will be saved upon closing our widget. In our widget, we won't be setting these variables directly, but will instead use them in conjunction with GUI controls.

3.2 Controls and module *gui*

We will use the `Orange.widgets.gui` to create/define the gui. With this library, the GUI definition part of the options box is a bit dense but rather very short

```
        box = gui.widgetBox(self.controlArea, "Info")
        self.infoa = gui.widgetLabel(box, 'No data on input yet, waiting to get␣
↪something.')
        self.infob = gui.widgetLabel(box, '')

        gui.separator(self.controlArea)
        self.optionsBox = gui.widgetBox(self.controlArea, "Options")
        gui.spin(self.optionsBox, self, 'proportion',
                 minv=10, maxv=90, step=10, label='Sample Size [%]:',
                 callback=[self.selection, self.checkCommit])
        gui.checkBox(self.optionsBox, self, 'commitOnChange',
                     'Commit data on selection change')
```

(continues on next page)

(continued from previous page)

```
        gui.button(self.optionsBox, self, "Commit", callback=self.commit)
        self.optionsBox.setDisabled(True)
```

We are already familiar with the first part - the Info group box. To make widget nicer, we put a separator between this and Options box. After defining the option box, here is our first serious `gui` control: a `Orange.widgets.gui.spin()`. The first parameter specifies its parent widget/layout, in this case `self.optionsBox` (the resulting widget object will automatically append itself to the parent's layout). The second (`self`) and third (`'proportion'`) define the *property binding* for the spin box. I.e. any change in the spin box control will automatically be propagated to the `self.proportions` and vice versa - changing the value of *self.proprotions* in the widget code by assignment (e.g. `self.proprotions = 30`) will update the spin box's state to match.

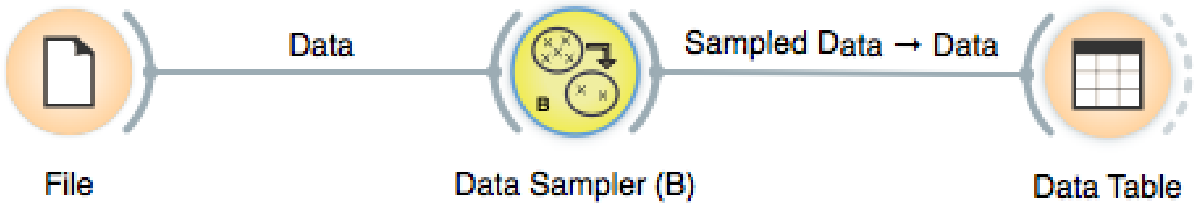
The rest of the spin box call gives some parameters for the control (minimum and maximum value and the step size), tells about the label which will be placed on the top, and tells it which functions to call when the value in the spin box is changed. We need the first callback to make a data sample and report in the Info box what is the size of the sample, and a second callback to check if we can send this data out. In `Orange.widgets.gui`, callbacks are either references to functions, or a list with references, just like in our case.

With all of the above, the parameters for the call of `Orange.widgets.gui.checkBox()` should be clear as well. Notice that this and a call to `Orange.widgets.gui.spin()` do not need a parameter which would tell the control the value for initialization: upon construction, both controls will be set to the value that is pertained in the associated setting variable.

That's it. Notice though that we have, as a default, disabled all the controls in the Options box. This is because at the start of the widget, there is no data to sample from. But this also means that when process the input tokens, we should take care for enabling and disabling. The data processing and token sending part of our widget now is

```
    @Inputs.data
    def set_data(self, dataset):
        if dataset is not None:
            self.dataset = dataset
            self.infoa.setText('%d instances in input dataset' % len(dataset))
            self.optionsBox.setDisabled(False)
            self.selection()
        else:
            self.dataset = None
            self.sample = None
            self.optionsBox.setDisabled(False)
            self.infoa.setText('No data on input yet, waiting to get something.')
            self.infob.setText('')
        self.commit()

    def selection(self):
        if self.dataset is None:
            return

        n_selected = int(numpy.ceil(len(self.dataset) * self.proportion / 100.))
        indices = numpy.random.permutation(len(self.dataset))
        indices = indices[:n_selected]
        self.sample = self.dataset[indices]
        self.infob.setText('%d sampled instances' % len(self.sample))

    def commit(self):
        self.Outputs.sample.send(self.sample)

    def checkCommit(self):
        if self.commitOnChange:
            self.commit()
```

You can now also inspect the `complete code` of this widget. To distinguish it with a widget we have developed in the previous section, we have designed a special `icon` for it. If you wish to test this widget in the Orange Canvas, put its code in the *orangedemo* directory we have created for the previous widget and try it out using a schema with a File and Data Table widget.



Well-behaved widgets remember their settings - the state of their checkboxes and radio-buttons, the text in their line edits, the selections in their combo boxes and similar.

3.3 Persisting defaults

When a widget is removed, its settings are stored to be used as defaults for future instances of this widget.

Updated defaults are stored in user's profile. It's location depends on the operating system: (%APPDATA%Orange<version>widgets on windows, ~/Library/ApplicationSupport/orange/<version>/widgets on macOS, ~/.local/share/Orange/<version>/widgets on linux) Original default values can be restored by deleting files from this folder, by running Orange from command line with *–clear-widget-settings* option, or through Options/Reset Widget Settings menu action.

3.3.1 Schema-only settings

Some settings have defaults that should not change. For instance, when using a Paint Data widget, drawn points should be saved in a workflow, but a new widget should always start with a blank page - modified value should not be remembered.

This can be achieved by declaring a setting with a *schema_only* flag. Such setting is saved with a workflow, but its default value never changes.

3.4 Context dependent settings

Context dependent settings are settings which depend on the widget's input. For instance, the scatter plot widget contains settings that specify the attributes for x and y axis, and the settings that define the color, shape and size of the examples in the graph.

An even more complicated case is the widget for data selection with which one can select the examples based on values of certain attributes. Before applying the saved settings, these widgets needs to check their compliance with the domain of the actual data set. To be truly useful, context dependent settings needs to save a setting configuration for each particular dataset used. That is, when given a particular dataset, it has to select the saved settings that is applicable and matches best currently used dataset.

Saving, loading and matching contexts is taken care of by context handlers. Currently, there are only two classes of context handlers implemented. The first one is the abstract `ContextHandler` and the second one is `DomainContextHandler` in which the context is defined by the dataset domain and where the settings contain attribute names. The latter should cover most of your needs, while for more complicated widgets you will need to derive a new classes from it. There may even be some cases in which the context is not defined by the domain, in which case the `ContextHandler` will be used as a base for your new handler.

Contexts need to be declared, opened and closed. Opening and closing usually takes place (in the opposite order) in the function that handles the data signal. This is how it looks in the scatter plot (the code is somewhat simplified for clarity).

```
@Input.data
def set_data(self, data):
    self.closeContext()
    self.data = data
    self.graph.setData(data)

    self.initAttrValues()

    if data is not None:
        self.openContext(data.domain)

    self.updateGraph()
    self.sendSelections()
```

In general, the function should go like this:

- Do any clean-up you need, but without clearing any of the settings that need to be saved. Scatter plot needs none.
- Call `self.closeContext()`; this ensures that all the context dependent settings (e.g. attribute names from the list boxes) are remembered.
- Initialize the widget state and set the controls to some defaults as if there were no context retrieving mechanism. Scatter plot does it by calling `self.initAttrValues()` which assigns the first two attributes to the x and y axis and the class attribute to the color. At this phase, you shouldn't call any functions that depend on the settings, such as drawing the graph.
- Call `self.openContext(data.domain)` (more about the arguments later). This will search for a suitable context and assign the controls new values if one is found. If there is no saved context that can be used, a new context is created and filled with the default values that were assigned at the previous point.
- Finally, adjust the widget according to the retrieved controls. Scatter plot now plots the graph by calling `self.updateGraph()`.

When opening the context, we provide the arguments on which the context depends. In case of `DomainContextHandler`, which scatter plot uses, we can give it a `Orange.data.Domain`. Whether a saved context can be reused is judged upon the presence of attributes in the domain.

If the widget is constructed appropriately (that is, if it strictly uses `Orange.widgets.gui` controls instead of the Qt's), no other administration is needed to switch the context.

Except for declaring the context settings, that is. Scatter plot has this in its class definition

```
settingsHandler = DomainContextHandler()
attr_x = ContextSetting("")
attr_y = ContextSetting("")

auto_send_selection = Setting(True)
toolbar_selection = Setting(0)
color_settings = Setting(None)
selected_schema_index = Setting(0)
```

`settingsHandler = DomainContextHandler()` declares that Scatter plot uses `DomainContextHandler`. The `attr_x` and `attr_y` are declared as `ContextSetting`.

3.5 Migrations

At the beginning of this section of the tutorial, we created a widget with a setting called proportion, which contains a value between 0 and 100. But imagine that for some reason, we are not satisfied with the value any more and decide that the setting should hold a value between 0 and 1.

We update the setting's default value, modify the appropriate code and we are done. That is, until someone that has already used the old version of the widget open the new one and the widget crashes. Remember, when the widget is opened, it has the same settings that were used the last time.

Is there anything we can do, as settings are replaced with saved before the __init__ function is called? There is! Migrations to the rescue.

Widget has a special attribute called *settings_version*. All widgets start with a settings_version of 1. When incompatible changes are done to the widget's settings, its settings_version should be increased. But increasing the version by it self does not solve our problems. While the widget now knows that it uses different settings, the old ones are still broken and need to be updated before they can be used with the new widget. This can be accomplished by reimplementing widget's methods *migrate_settings* (for ordinary settings) and *migrate_context* (for context settings). Both method are called with the old object and the version of the settings stored with it.

If we bumped settings_version from 1 to 2 when we did the above mentioned change, our migrate_settings method would look like this:

```python
def migrate_settings(settings, version):
   if version < 2:
     if "proportion" in settings:
        settings["proportion"] = settings["proportion"] / 100
```

Your migration rules can be simple or complex, but try to avoid simply forgetting the values, as settings are also used in saved workflows. Imagine opening a complex workflow you have designed a year ago with the new version of Orange and finding out that all the settings are back to default. Not fun!

Warning: If you change the format of an existing setting in a backwards-incompatible way, you will also want to *change the name* of that setting. Otherwise, older versions of Orange won't be able to load workflows with the new setting format.

There are two helper functions you can use. `Orange.widget.settings.rename_settings(settings, old_name, new_name)` does the obvious operation on *settings*, which can be either a dictionary or a context, thus it can be called from *migrate_settings* or *migrate_context*.

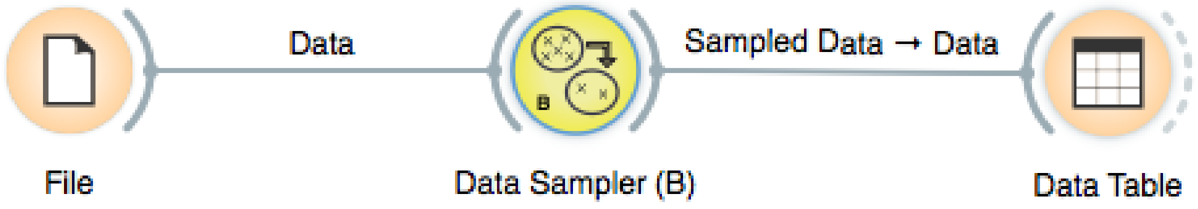
Another common operation may be upgrading your widget from storing variable names (as *str*) to storing variables (instances of classes derived from *Variable*). In a typical scenario, this happens when combo boxes are upgraded to using models. Function `Orange.widget.settings.migrate_str_to_variable(settings, names=None)` makes the necessary changes to the settings listed in *names*. *names* can be a list of setting names, a single string or *None*. In the latter case, all settings that may refer to variables (that is two-elements tuples constisting of a string and an int) are migrated.

So take some time, write the migrations and do not forget to bump the *settings_version* when you do breaking changes.

CHAPTER 4

Channels and Tokens

Our data sampler widget was, regarding the channels, rather simple and linear: the widget was designed to receive the token from one widget, and send an output token to another widget. Just like in an example schema below:

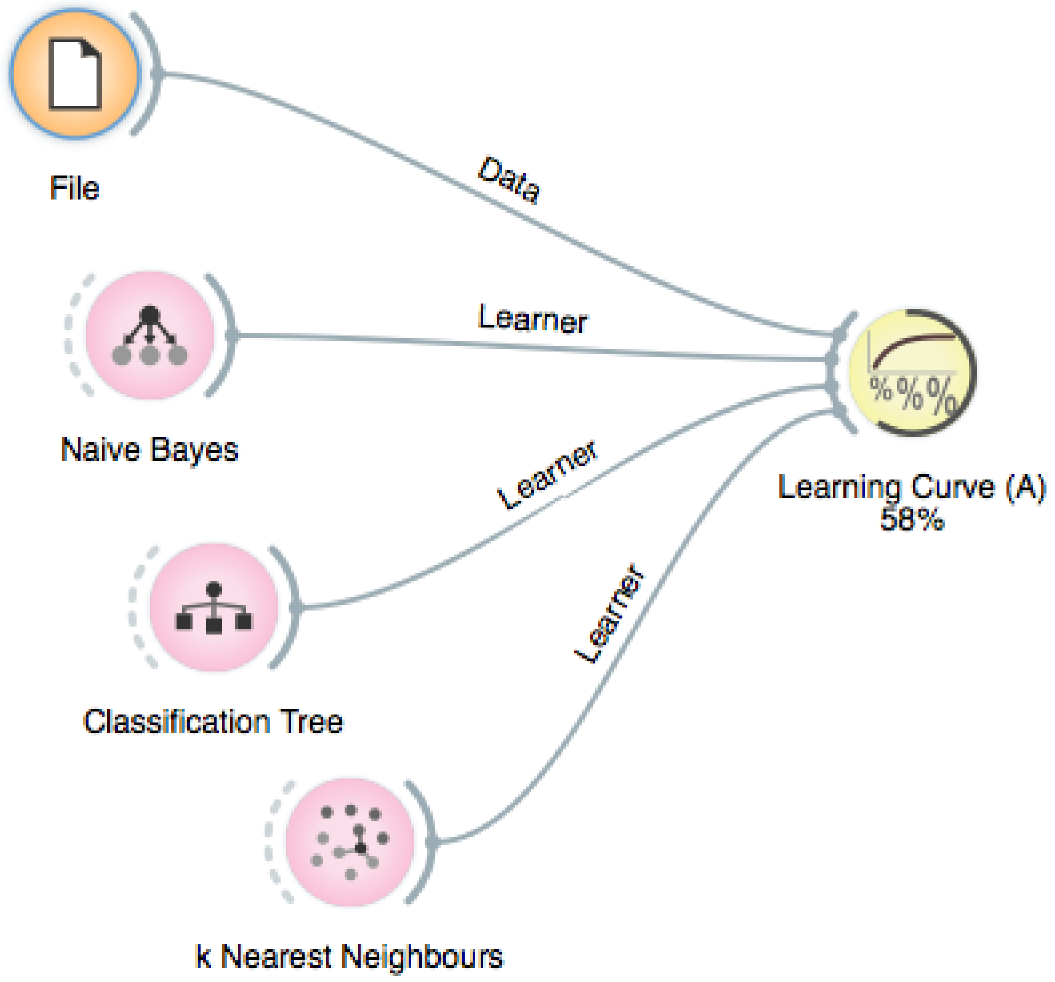


There's quite a bit more to channels and management of tokens, and we will overview most of the stuff you need to know to make your more complex widgets in this section.

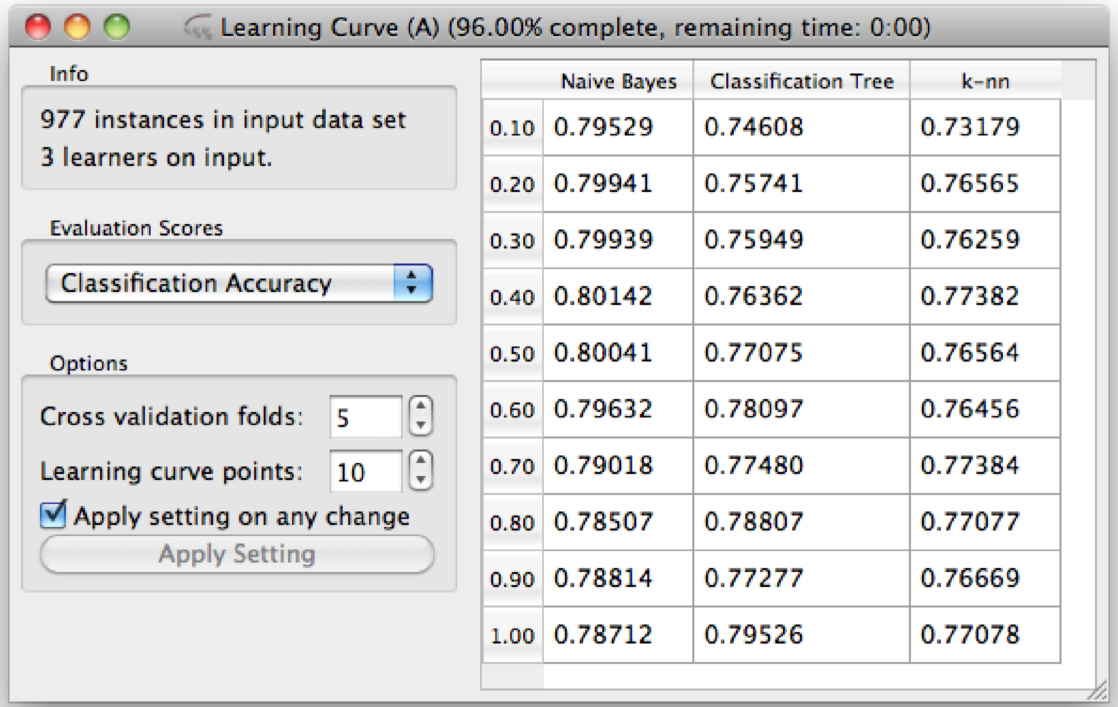
4.1 Multi-Input Channels

In essence, the basic idea about "multi-input" channels is that they can be used to connect them with several output channels. That is, if a widget supports such a channel, several widgets can feed their input to that widget simultaneously.

Say we want to build a widget that takes a dataset and test various predictive modeling techniques on it. A widget has to have an input data channel, and this we know how to deal with from our *previous* lesson. But, somehow differently, we want to connect any number of widgets which define learners to our testing widget. Just like in a schema below, where three different learners are used:



We will here take a look at how we define the channels for a learning curve widget, and how we manage its input tokens. But before we do it, just in brief: learning curve is something that you can use to test some machine learning algorithm in trying to see how its performance depends on the size of the training set size. For this, one can draw a smaller subset of data, learn the classifier, and test it on remaining dataset. To do this in a just way (by Salzberg, 1997), we perform k-fold cross validation but use only a proportion of the data for training. The output widget should then look something like:



977 instances in input data set
3 learners on input.

Classification Accuracy

Cross validation folds: 5

Learning curve points: 10

[✓] Apply setting on any change

	Naive Bayes	Classification Tree	k-nn
0.10	0.79529	0.74608	0.73179
0.20	0.79941	0.75741	0.76565
0.30	0.79939	0.75949	0.76259
0.40	0.80142	0.76362	0.77382
0.50	0.80041	0.77075	0.76564
0.60	0.79632	0.78097	0.76456
0.70	0.79018	0.77480	0.77384
0.80	0.78507	0.78807	0.77077
0.90	0.78814	0.77277	0.76669
1.00	0.78712	0.79526	0.77078

Now back to channels and tokens. Input and output channels for our widget are defined by

```
class Inputs:
    data = Input("Data", Orange.data.Table)
    learner = Input("Learner", Orange.classification.Learner,
                    multiple=True)
```

Notice that everything is pretty much the same as it was with widgets from previous lessons, the only difference being the additional argument `multiple=True`, which says that this input can be connected to outputs of multiple widgets.

Handlers of multiple-input signals must accept two arguments: the sent object and the id of the sending widget.

```
@Inputs.learner
def set_learner(self, learner, id):
    """Set the input learner for channel id."""
    if id in self.learners:
        if learner is None:
            # remove a learner and corresponding results
            del self.learners[id]
            del self.results[id]
            del self.curves[id]
        else:
            # update/replace a learner on a previously connected link
            self.learners[id] = learner
            # invalidate the cross-validation results and curve scores
            # (will be computed/updated in `_update`)
            self.results[id] = None
            self.curves[id] = None
```

(continues on next page)

(continued from previous page)

```
        else:
            if learner is not None:
                self.learners[id] = learner
                # initialize the cross-validation results and curve scores
                # (will be computed/updated in `_update`)
                self.results[id] = None
                self.curves[id] = None

        if len(self.learners):
            self.infob.setText("%d learners on input." % len(self.learners))
        else:
            self.infob.setText("No learners.")

        self.commitBtn.setEnabled(len(self.learners))
```

OK, this looks like one long and complicated function. But be patient! Learning curve is not the simplest widget there is, so there's some extra code in the function above to manage the information it handles in the appropriate way. To understand the signals, though, you should only understand the following. We store the learners (objects that learn from data) in an `OrderedDict` `self.learners`. This dictionary is a mapping of input *id* to the input value (the input learner itself). The reason this is an `OrderedDict` is that the order of the input learners is important as we want to maintain a consistent column order in the table view of the learning curve point scores.

The function above first checks if the channel *id* is already in `self.learners` and if so either deletes the corresponding entry if `learner` is `None` (remember receiving a `None` value means the link was removed/closed) or invalidates the cross validation results and curve point for that channel id, marking for update in `handleNewSignals()`. A similar case is when we receive a learner for a new channel id.

Note that in this widget the evaluation (k-fold cross validation) is carried out just once given the learner, dataset and evaluation parameters, and scores are then derived from class probability estimates as obtained from the evaluation procedure. Which essentially means that switching from one to another scoring function (and displaying the result in the table) takes only a split of a second. To see the rest of the widget, check out `its code`.

4.2 Using Several Output Channels

There's nothing new here, only that we need a widget that has several output channels of the same type to illustrate the idea of the default channels in the next section. For this purpose, we will modify our sampling widget as defined in previous lessons such that it will send out the sampled data to one channel, and all other data to another channel. The corresponding channel definition of this widget is

```
    class Outputs:
        sample = Output("Sampled Data", Orange.data.Table)
        other = Output("Other Data", Orange.data.Table)
```

We used this in the third incarnation of `data sampler widget`, with essentially the only other change in the code in the `selection()` and `commit()` functions

```
    def selection(self):
        if self.dataset is None:
            return

        n_selected = int(numpy.ceil(len(self.dataset) * self.proportion / 100.))
        indices = numpy.random.permutation(len(self.dataset))
        indices_sample = indices[:n_selected]
        indices_other = indices[n_selected:]
```

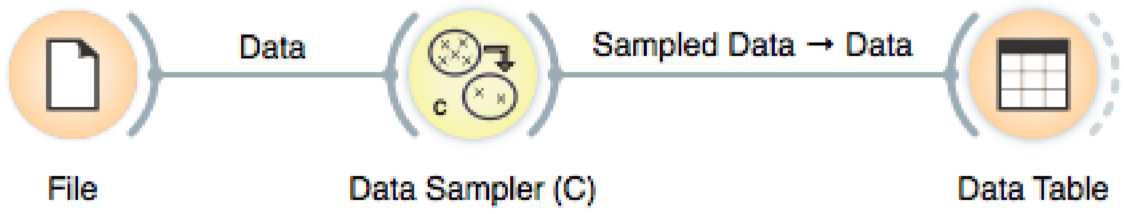
(continues on next page)

(continued from previous page)

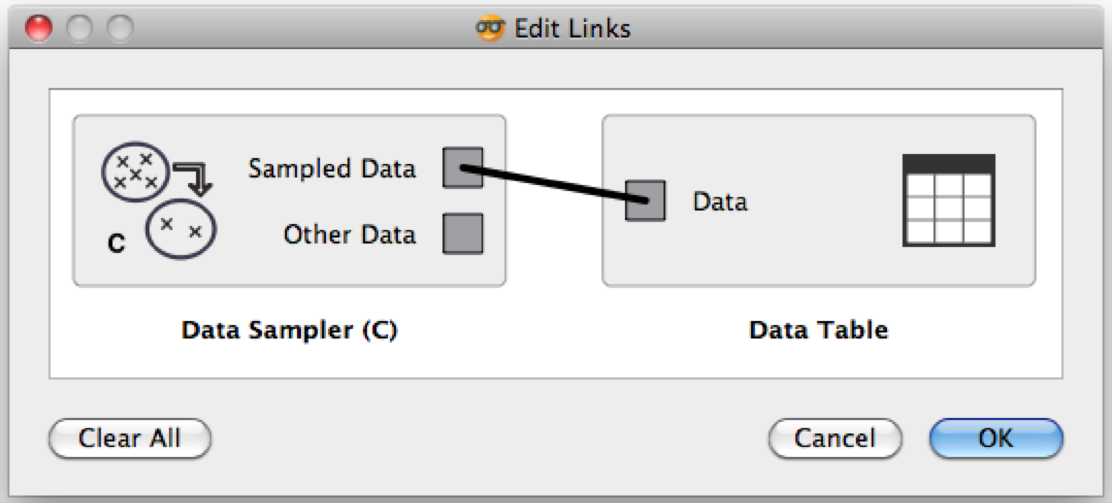
```
        self.sample = self.dataset[indices_sample]
        self.otherdata = self.dataset[indices_other]
        self.infob.setText('%d sampled instances' % len(self.sample))
```

```
    def commit(self):
        self.Outputs.sample.send(self.sample)
        self.Outputs.sample.send(self.otherdata)
```

If a widget that has multiple channels of the same type is connected to a widget that accepts such tokens, Orange Canvas opens a window asking the user to confirm which channels to connect. Hence, if we have just connected *Data Sampler (C)* widget to a Data Table widget in a schema below:



we would get a following window querying users for information on which channels to connect:



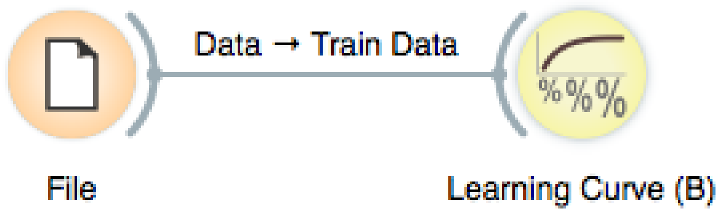
4.3 Default Channels (When Using Input Channels of the Same Type)

Now, let's say we want to extend our learning curve widget such that it does the learning the same way as it used to, but can - provided that such dataset is defined - test the learners (always) on the same, external dataset. That is, besides the training dataset, we need another channel of the same type but used for training dataset. Notice, however, that most often we will only provide the training dataset, so we would not like to be bothered (in Orange Canvas) with the dialog which channel to connect to, as the training dataset channel will be the default one.

When enlisting the input channel of the same type, the default channels have a special flag in the channel specification list. So for our new `learning curve` widget, the channel specification is

```
    class Inputs:
        data = Input("Data", Orange.data.Table, default=True)
        test_data = Input("Test Data", Orange.data.Table)
        learner = Input("Learner", Orange.classification.Learner,
                        multiple=True)
```

That is, the `Train Data` channel is a single-token channel which is a default one (third parameter). Note that the flags can be added (or OR-d) together so `Default + Multiple` is a valid flag. To test how this works, connect a file widget to a learning curve widget and - nothing will really happen:



That is, no window with a query on which channels to connect to will open, as the default *"Train Data"* was selected.

4.4 Explicit Channels

Sometimes when a widget has multiple outputs of different types, some of them should not be subject to this automatic default connection selection. An example of this is in Orange's *Logistic Regression* widget that outputs a supplementary 'Coefficients' data table. Such outputs can be marked with and `Explicit` flag, which ensures they are never selected for a default connection.

CHAPTER 5

Responsive GUI

And now for the hard part of making the widget responsive. We will do this by offloading the learner evaluations into a separate thread.

First read up on threading basics in Qt and in particular the subject of threads and qobjects and how they interact with the Qt's event loop.

We must also take special care that we can cancel/interrupt our task when the user changes algorithm parameters or removes the widget from the canvas. For that we use a strategy known as cooperative cancellation where we 'ask' the pending task to stop executing (in the GUI thread), then in the worker thread periodically check (at known predetermined points) whether we should continue, and if not return early (in our case by raising an exception).

5.1 Setting up

We use `Orange.widgets.utils.concurrent.ThreadExecutor` for thread allocation/management (but could easily replace it with stdlib's `concurrent.futures.ThreadPoolExecutor`).

```
import concurrent.futures
from Orange.widgets.utils.concurrent import (
    ThreadExecutor, FutureWatcher, methodinvoke
)
```

We will reorganize our code to make the learner evaluation an explicit task as we will need to track its progress and state. For this we define a *Task* class.

```
class Task:
    """
    A class that will hold the state for an learner evaluation.
    """
    #: A concurrent.futures.Future with our (eventual) results.
    #: The OWLearningCurveC class must fill this field
    future = ...       # type: concurrent.futures.Future

```

(continues on next page)

(continued from previous page)

```
    #: FutureWatcher. Likewise this will be filled by OWLearningCurveC
    watcher = ...      # type: FutureWatcher

    #: True if this evaluation has been cancelled. The OWLearningCurveC
    #: will setup the task execution environment in such a way that this
    #: field will be checked periodically in the worker thread and cancel
    #: the computation if so required. In a sense this is the only
    #: communication channel in the direction from the OWLearningCurve to the
    #: worker thread
    cancelled = False  # type: bool

    def cancel(self):
        """
        Cancel the task.

        Set the `cancelled` field to True and block until the future is done.
        """
        # set cancelled state
        self.cancelled = True
        # cancel the future. Note this succeeds only if the execution has
        # not yet started (see `concurrent.futures.Future.cancel`) ..
        self.future.cancel()
        # ... and wait until computation finishes
        concurrent.futures.wait([self.future])
```

In the widget's `__init__` we create an instance of the *ThreadExector* and initialize the task field.

```
        #: The current evaluating task (if any)
        self._task = None   # type: Optional[Task]
        #: An executor we use to submit learner evaluations into a thread pool
        self._executor = ThreadExecutor()
```

All code snippets are from `OWLearningCurveC.py`.

5.2 Starting a task in a thread

In *handleNewSignals* we call *_update*.

```
    def handleNewSignals(self):
        self._update()
```

And finally the *_update* function (from `OWLearningCurveC.py`) that will start/schedule all updates.

```
    def _update(self):
        if self._task is not None:
            # First make sure any pending tasks are cancelled.
            self.cancel()
        assert self._task is None

        if self.data is None:
            return
        # collect all learners for which results have not yet been computed
        need_update = [(id, learner) for id, learner in self.learners.items()
                       if self.results[id] is None]
```

(continues on next page)

(continued from previous page)

```
        if not need_update:
            return
```

At the start we cancel pending tasks if they are not yet completed. It is important to do this, we cannot allow the widget to schedule tasks and then just forget about them. Next we make some checks and return early if there is nothing to be done.

Continue by setting up the learner evaluations as a partial function capturing the necessary arguments:

```
        learners = [learner for _, learner in need_update]
        # setup the learner evaluations as partial function capturing
        # the necessary arguments.
        if self.testdata is None:
            learning_curve_func = partial(
                learning_curve,
                learners, self.data, folds=self.folds,
                proportions=self.curvePoints,
            )
        else:
            learning_curve_func = partial(
                learning_curve_with_test_data,
                learners, self.data, self.testdata, times=self.folds,
                proportions=self.curvePoints,
            )
```

Setup the task state and the communication between the main and worker thread. The only *state* flowing from the GUI to the worker thread is the *task.cancelled* field which is a simple trip wire causing the *learning_curve*'s callback argument to raise an exception. In the other direction we report the *percent* of work done.

```
        # setup the task state
        self._task = task = Task()
        # The learning_curve[_with_test_data] also takes a callback function
        # to report the progress. We instrument this callback to both invoke
        # the appropriate slots on this widget for reporting the progress
        # (in a thread safe manner) and to implement cooperative cancellation.
        set_progress = methodinvoke(self, "setProgressValue", (float,))

        def callback(finished):
            # check if the task has been cancelled and raise an exception
            # from within. This 'strategy' can only be used with code that
            # properly cleans up after itself in the case of an exception
            # (does not leave any global locks, opened file descriptors, ...)
            if task.cancelled:
                raise KeyboardInterrupt()
            set_progress(finished * 100)

        # capture the callback in the partial function
        learning_curve_func = partial(learning_curve_func, callback=callback)
```

See also:

`progressBarInit()`, `progressBarSet()`, `progressBarFinished()`

Next, we submit the function to be run in a worker thread and instrument a FutureWatcher instance to notify us when the task completes (via a *_task_finished* slot).

```
        self.progressBarInit()
        # Submit the evaluation function to the executor and fill in the
        # task with the resultant Future.
        task.future = self._executor.submit(learning_curve_func)
        # Setup the FutureWatcher to notify us of completion
        task.watcher = FutureWatcher(task.future)
        # by using FutureWatcher we ensure `_task_finished` slot will be
        # called from the main GUI thread by the Qt's event loop
        task.watcher.done.connect(self._task_finished)
```

For the above code to work, the *setProgressValue* needs defined as a pyqtSlot.

```
    @pyqtSlot(float)
    def setProgressValue(self, value):
        assert self.thread() is QThread.currentThread()
        self.progressBarSet(value)
```

5.3 Collecting results

In *_task_finished* (from `OWLearningCurveC.py`) we handle the completed task (either success or failure) and then update the displayed score table.

```
    @pyqtSlot(concurrent.futures.Future)
    def _task_finished(self, f):
        """
        Parameters
        ----------
        f : Future
            The future instance holding the result of learner evaluation.
        """
        assert self.thread() is QThread.currentThread()
        assert self._task is not None
        assert self._task.future is f
        assert f.done()

        self._task = None
        self.progressBarFinished()

        try:
            results = f.result()  # type: List[Results]
        except Exception as ex:
            # Log the exception with a traceback
            log = logging.getLogger()
            log.exception(__name__, exc_info=True)
            self.error("Exception occurred during evaluation: {!r}"
                       .format(ex))
            # clear all results
            for key in self.results.keys():
                self.results[key] = None
        else:
            # split the combined result into per learner/model results ...
            results = [list(Results.split_by_model(p_results))
                       for p_results in results]  # type: List[List[Results]]
            assert all(len(r.learners) == 1 for r1 in results for r in r1)
            assert len(results) == len(self.curvePoints)
```

(continues on next page)

(continued from previous page)

```
            learners = [r.learners[0] for r in results[0]]
            learner_id = {learner: id_ for id_, learner in self.learners.items()}

            # ... and update self.results
            for i, learner in enumerate(learners):
                id_ = learner_id[learner]
                self.results[id_] = [p_results[i] for p_results in results]
```

5.4 Stopping

Also of interest is the *cancel* method. Note that we also disconnect the *_task_finished* slot so that *_task_finished* does not receive stale results.

```
    def cancel(self):
        """
        Cancel the current task (if any).
        """
        if self._task is not None:
            self._task.cancel()
            assert self._task.future.done()
            # disconnect the `_task_finished` slot
            self._task.watcher.done.disconnect(self._task_finished)
            self._task = None
```

We also use cancel in `onDeleteWidget()` to stop if/when the widget is removed from the canvas.

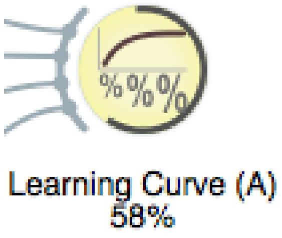
```
    def onDeleteWidget(self):
        self.cancel()
        super().onDeleteWidget()
```

CHAPTER 6

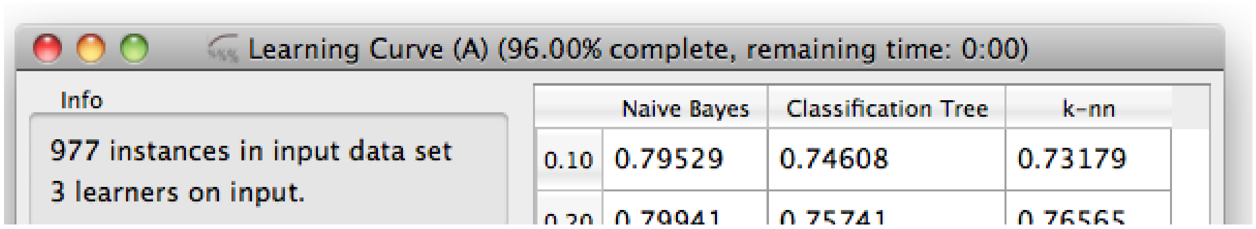
Utilities

6.1 Progress Bar

Operations that take more than a split second indicate their progress with a progress bar



and in the title bar of the widget's window.



977 instances in input data set
3 learners on input.

	Naive Bayes	Classification Tree	k-nn
0.10	0.79529	0.74608	0.73179
0.20	0.79941	0.75741	0.76565

There are two mechanisms for implementing this.

6.1.1 Progress bar class

Class *Orange.widgets.gui.ProgressBar* is initialized with a widget and the number of iterations:

```
progress = Orange.widgets.gui.ProgressBar(self, n)
```

6.1.2 Direct manipulation of progress bar

The progress bar can be manipulated directly through functions

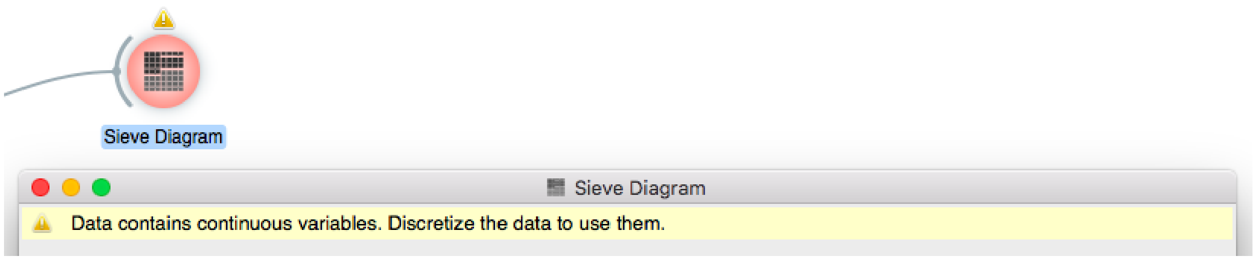
- `progressBarInit(self)`
- `progressBarSet(self, p)`
- `progressBarFinished(self)`

progressBarInit initializes the progress bar, *progressBarSet* sets it to *p* percents (from 0 to 100), and *progressBarFinished* closes it.

The code that uses these methods must use a try-except or try-finally block to ensure that the progress bar is removed if the exception is raised.

6.2 Issuing warning and errors

Widgets can show information messages, warnings and errors. These are displayed in the top row of the widget and also indicated in the schema.



Data contains continuous variables. Discretize the data to use them.

6.2.1 Simple messages

If the widget only issues a single error, warning and/or information at a time, it can do so by calling *self.error(text, shown=True)*, *self.warning(text, shown=True)* or *self.information(text, shown=True)*. Multiple messages - but just one of each kind - can be present at the same time:

```
self.warning("Discrete features are ignored.")
self.error("Fitting failed due to missing data.")
```

At this point, the widget has a warning and an error message.

> self.error("Fitting failed due to weird data.")

This replaces the old error message, but the warning remains.

The message is removed by setting an empty message, e.g. *self.error()*. To remove all messages, call *self.clear_messages()*.

If the argument *shown* is set to *False*, the message is removed:

```
self.error("No suitable features", shown=not self.suitable_features)
```

"Not showing" a message in this way also remove any existing messages.

6.2.2 Multiple messages

Widget that issue multiple independent messages that can appear simultaneously, need to declare them within local classes within the widget class, and derive them from the corresponding *OWWidget* classes for a particular kind of a message. For instance, a widget class can contain the following classes:

```
class Error(OWWidget.Error):
    no_continuous_features = Msg("No continuous features")

class Warning(OWWidget.Warning):
    empty_data = Msg("Comtrongling does not work on meta data")
    no_scissors_run = Msg("Do not run with scissors")
    ignoring_discrete = Msg("Ignoring {n} discrete features: {}")
```

Within the widget, errors are raised via calls like:

```
self.Error.no_continuous_features()
self.Warning.no_scissors_run()
```

As for the simpler messages, the *shown* argument can be added:

```
self.Warning.no_scissors_run(shown=self.scissors_are_available)
```

If the message includes formatting, the call must include the necessary data for the *format* method:

```
self.Warning.ignoring_discrete(", ".join(attrs), n=len(attr))
```

Message is cleared by:

```
self.Warning.ignoring_discrete.clear()
```

Multiple messages can be removed as in the simpler schema, with:

```
self.Warning.clear()
```

or:

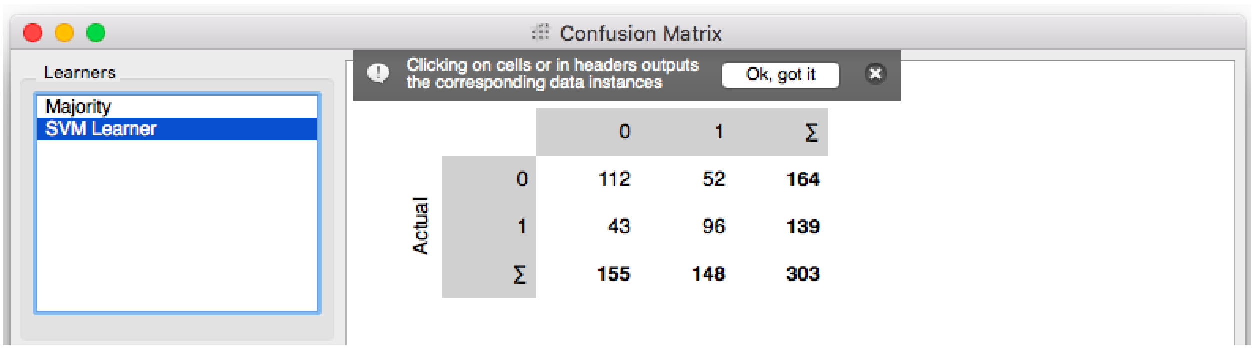
```
self.clear_messages()
```

Messages of both kinds - those from messages classes and those issued by, for instance, *self.error* - can coexist. Note, though, that methods for removing all messages of certain type (e.g. *self.Error.clear()*) or all messags (*self.clear_message()*) apply to all messages of this type.

Note: handling multiple messages through ids, that is, using *self.information(id, text)*, *self.warning(id, text)* and *self.error(id, text)* is deprecated and will be removed in the future.

6.3 Tips

Widgets can provide tips about features that are not be obvious or exposed in the GUI.



Clicking on cells or in headers outputs the corresponding data instances

		0	1	Σ
Actual	0	112	52	**164**
	1	43	96	**139**
	Σ	**155**	**148**	**303**

Such messages are stored in widget's class attribute *UserAdviceMessages*. When a widget is first shown, a message from this list is selected for display. If a user accepts (clicks 'Ok. Got it') the choice is recorded and the message is never shown again; just closing the message will not mark it as seen. Messages can be displayed again by pressing Shift + F1.

UserAdviceMessages contains instances of `Message`. The messages contains a text and an id (also a string), and, optionally, an icon and an URL with further information.

The confusion matrix widget sets up the following list:

```
UserAdviceMessages = [
    widget.Message("Clicking on cells or in headers outputs the "
                       "corresponding data instances",
                   "click_cell")]
```

CHAPTER 7

OWWidget

The `OWWidget` is the main component for implementing a widget in the Orange Canvas workflow. It both defines the widget input/output capabilities and implements it's functionality within the canvas.

7.1 Widget Meta Description

Every widget in the canvas framework needs to define it's meta definition. This includes the widget's name and text descriptions but more importantly also its input/output specification.

```
class IntConstant(OWWidget):
    name = "Integer Constant"
    description = "A simple integer constant"

    class Outputs:
        constant = Output("Constant", int)

    ...

    def commit(self):
        """Commit/send the outputs."""
        self.Outputs.constant.send(42)
```

Omitting the implementation details, this defines a simple node named *Integer Constant* which outputs (on a signal called *Constant*) a single object of type `int`.

The node's inputs are defined similarly. Each input is then used as a decorator of its corresponding handler method, which accepts the inputs at runtime:

```
class Adder(OWWidget):
    name = "Add two integers"
    description = "Add two numbers"

    class Inputs:
```

(continues on next page)

(continued from previous page)

```
        a = Input("A", int)
        b = Input("B", int)

    class Outputs:
        sum = Input("A + B", int)

    ...

    @Inputs.a
    def set_A(self, a):
        """Set the `A` input."""
        self.A = a

    @Inputs.b
    def set_B(self, b):
        """Set the `B` input."""
        self.B = b

    def handleNewSignals(self):
        """Coalescing update."""
        self.commit()

    def commit(self):
        """Commit/send the outputs"""
        sef.Outputs.sum.send("self.A + self.B)
```

See also:

Getting Started Tutorial

7.2 Input/Output Signal Definitions

Widgets specify their input/output capabilities in their class definitions by defining classes named *Inputs* and *Outputs*, which contain class attributes of type *Input* and *Output*, correspondingly. *Input* and *Output* require at least two arguments, the signal's name (as shown in canvas) and type. Optional arguments further define the behaviour of the signal.

Note: old-style signals define the input and output signals using class attributes *inputs* and *outputs* instead of classes *Input* and *Output*. The two attributes contain a list of tuples with the name and type and, for inputs, the name of the handler method. The optional last argument is an integer constant giving the flags. This style of signal definition is deprecated.

7.3 Sending/Receiving

The widgets receive inputs at runtime with the handler method decorated with the signal, as shown in the above examples.

If an input is defined with the flag *multiple* set, the input handler method also receives a connection *id* uniquely identifying a connection/link on which the value was sent (see also *Channels and Tokens*)

The widget sends an output by calling the signal's *send* method, as shown above.

7.4 Accessing Controls though Attribute Names

The preferred way for constructing the user interface is to use functions from module `Orange.widgets.gui` that insert a Qt widget and establish the signals for synchronization with the widget's attributes.

> gui.checkBox(box, self, "binary_trees", "Induce binary tree")

This inserts a *QCheckBox* into the layout of *box*, and make it reflect and changes the attriubte *self.binary_trees*. The instance of *QCheckbox* can be accessed through the name it controls. E.g. we can disable the check box by calling

> self.controls.binary_trees.setDisabled(True)

This may be more practical than having to store the attribute and the Qt widget that controls it, e.g. with

> **self.binarization_cb = gui.checkBox(** box, self, "binary_trees", "Induce binary tree")

7.5 Class Member Documentation

CHAPTER 8

Library of Common GUI Controls

`gui` is a library of functions which allow constructing a control (like check box, line edit or a combo), inserting it into the parent's layout, setting tooltips, callbacks and so forth, establishing synchronization with a Python object's attribute (including saving and retrieving when the widgets is closed and reopened) … in a single call.

Almost all functions need three arguments:

- the *widget* into which the control is inserted,
- the *master* widget with one whose attributes the control's value is synchronized,
- the name of that attribute (*value*).

All other arguments should be given as keyword arguments for clarity and also for allowing the potential future compatibility-breaking changes in the module. Several arguments that are common to all functions must always be given as keyword arguments.

8.1 Common options

All controls accept the following arguments that can only be given as keyword arguments.

tooltip A string for a tooltip that appears when mouse is over the control.

disabled Tells whether the control be disabled upon the initialization.

addSpace Gives the amount of space that is inserted after the control (or the box that encloses it). If *True*, a space of 8 pixels is inserted. Default is 0.

addToLayout The control is added to the parent's layout unless this flag is set to *False*.

stretch The stretch factor for this widget, used when adding to the layout. Default is 0.

sizePolicy The size policy for the box or the control.

8.2 Common Arguments

Many functions share common arguments.

widget Widget on which control will be drawn.

master Object which includes the attribute that is used to store the control's value; most often the *self* of the widget into which the control is inserted.

value String with the name of the master's attribute that synchronizes with the the control's value..

box Indicates if there should be a box that around the control. If `box` is *False* (default), no box is drawn; if it is a string, it is also used as the label for the box's name; if `box` is any other true value (such as `True` :), an unlabeled box is drawn.

callback A function that is called when the state of the control is changed. This can be a single function or a list of functions that will be called in the given order. The callback function should not change the value of the controlled attribute (the one given as the `value` argument described above) to avoid a cycle (a workaround is shown in the description of `listBox` function.

label A string that is displayed as control's label.

labelWidth The width of the label. This is useful for aligning the controls.

orientation When the label is given used, this argument determines the relative placement of the label and the control. Label can be above the control, (*"vertical"* or *True* - this is the default) or in the same line with control, (*"horizontal"* or *False*). Orientation can also be an instance of `QLayout`.

8.3 Properties

PyQt support settings of properties using keyword arguments. Functions in this module mimic this functionality. For instance, calling

cb = gui.comboBox(. . . , editable=True)

has the same effect as

cb = gui.comboBox(. . .) cb.setEditable(True)

Any properties that have the corresponding setter in the underlying Qt control can be set by using keyword arguments.

8.4 Common Attributes

box If the constructed widget is enclosed into a box, the attribute *box* refers to the box.

8.5 Widgets

This section describes the wrappers for controls like check boxes, buttons and similar. Using them is preferred over calling Qt directly, for convenience, readability, ease of porting to newer versions of Qt and, in particular, because they set up a lot of things that happen in behind.

8.6 Other widgets

8.7 Utility functions

8.8 Internal functions and classes

This part of documentation describes some classes and functions that are used internally. The classes will likely maintain compatibility in the future, while the functions may be changed.

8.8.1 Wrappers for Qt classes

8.8.2 Wrappers for Python classes

8.8.3 Other functions

CHAPTER 9

Testing Widgets

Writing widget tests may seem a daunting task at first, as there are so many different functionalities a widget depends on while it operates. But as long as you keep in mind that widgets are independent of each other, you should be fine.

Whenever you feel that you need to construct a workflow to test the widget, stop, and think again. The only way widgets communicate with each other is through signals. If you need to construct a workflow to prepare the data and send it to the widget, prepare the data in code and pass it in. Same goes for outputs, get the value of the widget output and check that the assumptions about it hold.

In order to make your life easier, Orange provides a base class for unittest `WidgetTest`, which provides some helper methods.

9.1 Class Member Documentation

CHAPTER 10

API

10.1 `Orange.widgets.utils.concurrent`