# funq Documentation

***Release 1.1.1***

**jpages**

July 18, 2014

Contents

**funq** is a tool to write FUNctional tests for Qt applications using python.

It is licenced under the CeCILL v2.1 licence (very close to the GPL v2). See the LICENCE.txt file distributed in the sources for more information.

Look at the github repository to see how to install funq: https://github.com/parkouss/funq.

CHAPTER 1

# Tutorial

This section aims to show by the example how to setup first **funq** tests on a project.

## 1.1 Requirements

This tutorial is based on a sample QT application that you can find here: https://github.com/parkouss/funq/tree/master/server/player_tester.

This sample application must be compiled to an executable binary file, **player_tester**.

---

**Important:** Please keep in mind that funq-server currently does only works with QT4 - be careful to not compile player_tester with QT5 !

---

The two packages **funq** and **funq-server** must be installed. You can check that funq-server is installed by running:

```
funq -h
```

And that funq is installed with:

```
nosetests -h | grep 'with-funq'
```

## 1.2 Tests file tree

The file tree of a **funq** project is basically a folder that contains test files and a funq configuration file, **funq.conf**.

First, create a *tutorial* folder next to the **player_tester** binary file:

```
mkdir tutorial
cd tutorial
```

Now you have to write the configuration file **funq.conf**. Here is the most minimalistic configuration file:

```
[applitest]
executable = ../player_tester
```

## 1.3 Write your first test

You can now write your first test. Put the following content in a file called test_1.py:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
 --------- Module's documentation ---------
This file is part of the funq tutorial. It currently defines
one test method.
"""

from funq.testcase import FunqTestCase
import time

class MyTestCase(FunqTestCase):
    # identify the configuration
    __app_config_name__ = 'applitest'

    def test_my_first_test(self):
        """
         --------- Test method documentation ---------
        """
        # do nothing for 3 seconds
        time.sleep(3)
```

This file contains one test, that do nothing except wait for 3 seconds.

---

**Note:** The "applitest" configuration is described in the funq configuration file by the section of the same name.

---

---

**Note:** Subclass `funq.testcase.FunqTestCase` ensure that each test method will start the application and close it properly.

---

## 1.4 Test execution

Well done ! Let's run this first test. Type the following command:

```
nosetests --with-funq
```

One window must appear, and close after a few seconds. The output on the terminal must look like this:

```
.
----------------------------------------------------------------------
Ran 1 test in 3.315s

OK
```

---

**Note:** The option `--with-funq` given to nosetests allow to use the funq plugin that will read the configuration file and execute your tests.

---

---

**Note:** **nosetests** has multiples options to allow for example the generation of an xml file to format tests result. See **nosetests -h**.

---

And voilà! You have written and launched your first funq test. Now let's go a bit further by adding two tests and use an aliases file.

Let's first create the aliases file.

## 1.5 Aliases file

This file associate a name (an alias) to graphical objects identified by their path. This behavior allow to keep running the written tests even if the widgets moves in the tested application or if the code of the tested application is refactored.

**applitest.aliases** file:

```
# the main central widget
mainWidget = fenPrincipale::widgetCentral

# and some other widgets

btnTest = {mainWidget}::btnTest

dialog1 = fenPrincipale::QMessageBox
dialog1_label = {dialog1}::qt_msgbox_label
tableview = {mainWidget}::tableView
treeview = {mainWidget}::treeView
```

**Note:** This file support variables substitution by using brackets, allowing to avoid useless and dangerous copy/paste.

**Note:** Some aliases are predefined. See *Predifined graphical kit aliases*.

Now you need to modify the **funq.conf** configuration file to indicate the use of this aliases file. Add the following line in the **applitest** section:

```
aliases = applitest.aliases
```

## 1.6 Do something with the widgets

Let's write another tests in a new file, **test_widgets.py**:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
This file is part of the funq tutorial. It briefly shows widget
interaction.
"""

from funq.testcase import FunqTestCase
from funq.client import FunqError

class TestCase2(FunqTestCase):
    # identify the configuration
    __app_config_name__ = 'applitest'

    def test_libelle_btn_test(self):
        """
        Test the test button libelle
        """
        # use  the "btnTest" alias
        btn_test = self.funq.widget('btnTest')
        properties = btn_test.properties()
```

```
        self.assertEquals(properties['text'], 'Test')

    def test_open_dialog(self):
        """
        Test that a dialog is open when user click on the test button.
        """
        self.funq.widget('btnTest').click()
        dlg_label = self.funq.widget('dialog1_label')

        self.assertEquals(dlg_label.properties()['text'], "Button clicked")

    def test_tableview_content(self):
        """
        Test the data in tableview.
        """
        view = self.funq.widget('tableview')
        items = list(view.model_items().iter())
        self.assertEquals(len(items), 16)

        for item in items:
            text = "row {r}, column {c}".format(r=item.row,
                                                c=item.column)
            self.assertEquals(item.value, text)

    def test_some_treeview_content(self):
        """
        test some data in the treeview
        """
        model = self.funq.widget('treeview').model_items()

        item = model.item_by_named_path([u"item 1", u"item 1-2"])
        parent_item = model.item_by_named_path([u"item 1"])

        self.assertEquals(item.value, u"item 1-2")
        self.assertIn(item, parent_item.items)
```

You can see that the member variable **self.funq** is the entry point to manipulate the tested application. It is an instance of `funq.client.FunqClient`.

Now you can start tests again:

```
nosetests --with-funq
```

---

**Note:** This time, 5 tests are launched! It's normal because you have written 5 tests divided up in 2 files.

To launch the tests of one file only, name it on the command line:

```
nosetests --with-funq test_widgets.py
```

---

---

**Important:** It is really important even for functional tests to not write tests that depends on others tests. In other words, the *order of test execution must not be important*. This allow to limit side effects and to find quickly why a test failed. This being said, **nosetests** does not assure any order in test execution.

---

## 1.7 Going further

This is the end of this tutorial. To go further, you must look at the API documentation!

CHAPTER 2

# The *funq.conf* configuration file

This configuration file describe the execution tests environment.

It's an **ini** configuration file that contains sections (between [brackets]), and eah section is the configuration of one application to test.

Here is an example of a funq configuration file:

```
[applitest]
executable = ../player_tester
aliases = applitest.aliases

[deuxieme_appli]
executable = /path/to/my/program
args = arg1 "arg 2"
funq_port = 55000

[example_detache]
executable = socket://localhost
funq_port = 55001
```

The only required option in each section is **executable**. If its value starts with "socket://" then the application will not be launched but the tests will try to connect to the address instead. This is the **detached** mode, allowing to test an already launched application (note that this application will have to be started with the **funq** executable, or compiled with libFunq).

Here is the list of the facultative availables options:

- **args**: executable arguments
- **funq_port**: libFunq communication port used (défaut: 9999). May be 0, an in this case the OS will pick the first available port.
- **cwd**: path to the execution directory. By default, this is the executable directory.
- **aliases**: path to the aliases file.
- **executable_stdout**: a file to save stdout output. Can be null to not redirect output.
- **executable_stderr**: a file to save stderr output. Can be null to not redirect output.
- **timeout_connection**: timeout in seconds to try to connect with the libFunq socket. Defaults to 10 seconds.
- **attach**: set to "no" or "0" to enable the *detached mode* of funq. See *Compile the application with libFunq*.
- **with_valgrind**: set to "1" or "yes" to activate valgrind
- **valgrind_args**: valgrind arguments. By default, "–leak-check=full –show-reachable=yes".

- **screenshot_on_error**: set to "1" or "yes" to automatically take screenshot on tests errors. A *screenshot-errors* will then be created and will contains screnshots of failed tests.

CHAPTER 3

# Writing tests

## 3.1 Nomenclature

- Tests must be written in python files named **test*.py**
- Tests may be placed in folders like **test***
- Tests must be written as methods of subclasses of `funq.testcase.FunqTestCase` or `funq.testcase.MultiFunqTestCase`
- Tests methods must be named **test***.

Example:

*file test_something.py*:

```
from funq.testcase import FunqTestCase

class MyClass(object):
    """
    A standard class (not a test one).
    """

class MyTest(FunqTestCase):
    """
    A test class.
    """
    __app_config_name__ = 'my_conf'

    def test_something(self):
        """
        A test method.
        """

    def something(self):
        """
        A method that is not a test method.
        """
```

## 3.2 Using assert* methods

Inside test methods, it is highly recommended to use **assert*** methods to detect test failures.

The complete list of these methods is availables in the documentation for `unittest.TestCase`. Here are some of these methods:

- `unittest.TestCase.assertTrue()`
- `unittest.TestCase.assertFalse()`
- `unittest.TestCase.assertEqual()`
- `unittest.TestCase.assertNotEqual()`
- `unittest.TestCase.assertIs()`
- `unittest.TestCase.assertIsNot()`
- `unittest.TestCase.assertIn()`
- `unittest.TestCase.assertNotIn()`
- `unittest.TestCase.assertIsInstance()`
- `unittest.TestCase.assertNotIsInstance()`
- `unittest.TestCase.assertRegexpMatches()`
- `unittest.TestCase.assertRaises()`
- `unittest.TestCase.assertRaisesRegexp()`
- ...

Example:

```
from funq.testcase import FunqTestCase

class MyTest(FunqTestCase):
    __app_config_name__ = 'my_conf'

    def test_something(self):
        self.assertEqual(1, 1, "Error message")
```

## 3.3 Skipped or todo tests

It is useful to not start tests in some cases, or to mark them "uncomplete" (todo). For this, there are some decorators:

- `unittest.skip()`, `unittest.skipIf()`, `unittest.skipUnless()`
- `unittest.expectedFailure()`
- `funq.testcase.todo()`

Example:

```
from funq.testcase import FunqTestCase, todo
from unittest import skipIf
import sys

class MyTest(FunqTestCase):
    __app_config_name__ = 'ma_conf'

    @todo("Waiting for this to work !")
    def test_something(self):
        self.assertEqual(1, 2, "Error message")
```

```
    @skipIf(sys.platform.startswith("win"), "requires Windows")
    def test_other_thing(self):
        ....
```

## 3.4 Parameterized tests

**Funq** offer a way to generate test methods given a base method and some data. This works for methods of subclasses of `funq.testcase.BaseTestCase` (`funq.testcase.FunqTestCase` or `funq.testcase.MultiFunqTestCase`), and by using appropriate decorators:

- `funq.testcase.parameterized()`
- `funq.testcase.with_parameters()`

Example:

```
from funq.testcase import FunqTestCase, parameterized, with_parameters

PARAMS = [
    ('1', [1], {}),
    ('2', [2], {}),
]

class MyTest(FunqTestCase):
    __app_config_name__ = 'my_conf'

    @parameterized('2', 2)
    @parameterized('3', 3)
    def test_something(self, value):
        self.assertGreater(value, 1)

    @with_parameters(PARAMS)
    def test_other_thing(self, value):
        self.assertLess(0, value)
```

CHAPTER 4

# Launching tests (nose)

Tests are launched by nose, and you have to tell it to use the **funq** plugin.

## 4.1 Manual launching

Basically:

```
nosetests --with-funq
```

The command must be started from the folder containing tests files and the **funq.conf** configuration file.

---

**Note:** There are many options for nose, and some specifics to the **funq** plugin. See `nosetests --help` for an exhaustive list.

---

Example:

```
# launch tests with all stdout/stderr output and stop on the first error
nosetests --with-funq -s -x
```

## 4.2 Defining default options

Every nose option may be specified by default in a file named **setup.cfg**. You can look at the nose documentation for more informations.

Example:

```
[nosetests]
verbosity=2
with-funq=1
```

---

**Note:** This configuration is very useful, and allow to type only **nosetests** on the command line instead of `nosetests --with-funq -vv`. I highly recommend this configuration and I will use it in the following documentation.

---

## 4.3 Selecting tests to launch

It s possible to select tests to launch using nose.

Example:

```
# every tests in a given file
nosetests test_export.py

# every tests of a given class in a test file
nosetests test_export.py:TestExportElectre

# just one test (one method)
nosetests test_export.py:TestExportElectre.test_export_b6
```

---

**Note:** See the nose documeation fo more information.

If the verbosity option is equal to 2, the tests execution will show test names with the same format. This means that you can then copy/paste a test name to restart it.

---

## 4.4 Going further

**nose** got plenty of usefuls plugins !

Some are integrated in nose, others are third-party plugins and need a proper installation.

Som of the interesting nose plugins are listed here:

- **xunit**: format tests output using xunit
- **attributeselector**: select tests given their attributes
- **collect-only**: allow to only list tests without really execute them

See the nose documentation, and google to find others usefuls plugins !

---

**Note:** It is also easy to write your own nose plugins.

---

CHAPTER 5

# How to find widget's paths

Currently there is two ways fo ind widget's paths, and it is explained in this section.

## 5.1 With funq

The easiest way is to start **funq** executable (from *funq-server* package) in *pick mode*.

For example, to find widgets from qtcreator application:

```
funq --pick qtcreator
```

Then you need to pick on a widget while pressing *Ctrl* and *Shift*. This will print on stdout the complete widget path and the available properties.

Here is an example of output when clicking on the "File" menu in qtcreator:

```
WIDGET: `Core:_:Internal:_:MainWindow-0::QtCreator.MenuBar` (pos: 42, 12)
      objectName: QtCreator.MenuBar
      modal: false
      windowModality: 0
      enabled: true
      x: 0
      y: 0
      width: 1091
      height: 25
      minimumWidth: 0
      minimumHeight: 0
      maximumWidth: 16777215
      maximumHeight: 16777215
      font: Sans,10,-1,0,50,0,0,0,0,0
      mouseTracking: true
      isActiveWindow: true
      focusPolicy: 0
      focus: false
      contextMenuPolicy: 1
      updatesEnabled: true
      visible: true
      minimized: false
      maximized: false
      fullScreen: false
      acceptDrops: false
      windowOpacity: 1
      windowModified: false
```

```
    layoutDirection: 0
    autoFillBackground: false
    inputMethodHints: 0
    defaultUp: false
    nativeMenuBar: false
```

## 5.2 Xml dump of all widgets

It is also possible to dump widgets of the running application. This may only be used in a test:

```
from funq.testcase import FunqTestCase

class MyTestCase(FunqTestCase):
    __app_config_name__ = 'applitest'

    def test_my_first_test(self):
        # this will write a "dump.json" file
        self.funq.dump_widgets_list('dump.json')
```

CHAPTER 6

# Test multiple applications at the same time

It is possible to test multiple applications in one test method. For this you will have to write multiple sections in the **funq.conf** configuration file.

```
[applitest]
executable = ../player_tester
aliases = applitest.aliases
funq_port = 10000

[applitest2]
executable = /path/to/my/program
funq_port = 10001
```

**Important:** Be careful to specify different ports for each application !

Let's see the test code now:

```
from funq.testcase import MultiFunqTestCase

class MyTestCase(MultiFunqTestCase):
    # wanted configurations
    __app_config_names__ = ('applitest', 'applitest2')

    def test_my_first_test(self):
        # FunqClient objects will then be accessible with a dict:
        # - self.funq['applitest'] to interact with "applitest"
        # - self.funq['applitest2'] to interact with "applitest2"
        pass
```

**Note:** There is some differences when interacting with multiples applications at the same time:

```
- using :class:`funq.testcase.MultiFunqTestCase`
- using **__app_config_names__** instead of **__app_config_name__**
- **self.funq** becomes a dict where keys are configuration names
  and associated values are instances of :class:`funq.client.FunqClient`.
```

**Note:** The number testables applications at the same time is not limited.

CHAPTER 7

# Attach to an already started application

It is possible to **attach to an already started application** instead of letting funq start and shutdown the application.

For this you need to specify in the **funq.conf** configuration file "socket://" followed by the IP address in the "executable" field.

```
[applitest]
executable = socket://localhost
funq_port = 49000
```

**Important:** In this case, funq is not responsible for starting and stopping the tested application. This must be done somewhere else, with the libFunq server integrated.

CHAPTER 8

# Predifined graphical kit aliases

QT is naming some graphical objects differently given the window manager used. Some aliases are predefined to point to the same objects for multiple window managers.

Here is the file content that defines these aliases:

```
[default]
QT_FILE_DIALOG = QFileDialog
QT_FILE_DIALOG_LINE_EDIT = {QT_FILE_DIALOG}::fileNameEdit
QT_FILE_DIALOG_BTN_OK = {QT_FILE_DIALOG}::buttonBox::QPushButton

[kde]
QT_FILE_DIALOG = KFileDialog
QT_FILE_DIALOG_LINE_EDIT = {QT_FILE_DIALOG}::KFileWidget::KUrlComboBox::KLineEdit
QT_FILE_DIALOG_BTN_OK = {QT_FILE_DIALOG}::KFileWidget::KPushButton
```

Each section defines a particular graphical kit (window manager), **default** being the kit used by default.

You can use these aliases in the aliases file in a standard way (between {}).

**Note:** Currently the famework does not identify automatically which window manager is used - **default** is always used.

To use another graphical kit, you have to specify it with the *funq-gkit* nose option. Example:

```
nosetests --with-funq --funq-gkit kde
```

CHAPTER 9

# Compile the application with libFunq

**Note:** Always prefer the standard approach (use **funq** executable) when possible. This part of the documentation is only useful when the standard approach is not working. ne doit être utilisée que si la méthode standard pose problème.

**libFunq** can be integrated in an already compiled application with the code injection done by the **funq** executable. It is the preferred way since this does not require to modify the source code of the tested application.

But it is also possible to integrate directly libFunq in an application if you need it. You will then need to modify your .pro file like this:

```
include(/path/to/libFunq/libFunq.pri)
```

Then, in the main of your program:

```
#include "funq.h"
#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);

    // libFunq activation
    Funq::activate(true);

    /* ... */

    return a.exec();
}
```

You will then need to adapt the **funq.conf** configuration file:

```
[my_application]
executable = mon_executable

# does not use funq executable to inject libFunq
attach = no
```

Once integrated in the compiled application, libFunq becomes a **security hole** as it allows to interact with the application by using the integrated TCP server.

The environment variable **FUNQ_ACTIVATION** if defined to 1 starts the TCP server at applcation startup and will allow funq clients to interact with the application.

To bypass this constraint, it is recommended to use #define in your code to integrate libFunq only for testing purpose and not deliver to final users an application with funq included.

**Important:** The best alternative is to use the dynamic injection provided by the executable **funq** when possible.

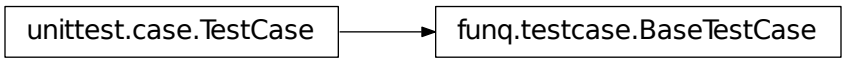
CHAPTER 10

# API of funq (client side)

## 10.1 TestCases and helpers

### 10.1.1 TestCases

Test classes in **funq** are subclasses of `unittest.TestCase`.




**class** `funq.testcase.`**`BaseTestCase`**(**args*, ***kwargs*)

Abstract class of a testcase for Funq.

It defines a common behaviour to name tests methods and uses the metaclass `MetaParameterized` that allows to generate methods from data.

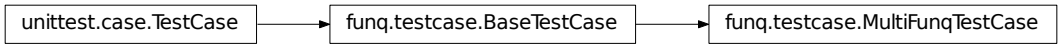
It inherits from `unittest.TestCase`, thus allowing to use very useful methods like assertEquals, assertFalse, etc.


**class** `funq.testcase.`**`FunqTestCase`**(**args*, ***kwargs*)

A testcase to launch an application and write tests against it.

The class attribute **__app_config_name__** is required and must contains the name of a section in the funq configuration file. A class attribute **__app_config__** will then be automatically created to give access to the configuration of the application (`funq.client.ApplicationConfig`).

**Variables** **funq** – instance of `funq.client.FunqClient`, allowing to manipulate the application.




**class** funq.testcase.**MultiFunqTestCase**(**args*, ***kwargs*)

A testcase to launch multiple applications at the same time and write tests against them.

The class attribute **__app_config_names__** is required and must contains a list of section's names in the funq configuration file. A class attribute **__app_configs__** will then be automatically created to give access to the configurations of the application (a dict with values of type funq.client.ApplicationConfig, where the keys are configuration names).

**Variables** **funq** – a dict that contains funq.client.FunqClient, allowing to manipulate the application. Keys are configuration names.

### 10.1.2 Helpers

funq.testcase.**todo**(*skip_message*, *exception_cls=<type 'exceptions.AssertionError'>*)

A decorator to skip a test on given exception types. If the decorated test pass, an exception AssertionSuccessError will be thrown.

It is possible to specify which type of Exception is handled with the **exception_cls** argument.

Example:

```
class MyTestCase(FunqTestCase):
    __app_config_name__ = 'ma_conf'

    @todo("this test needs something to pass")
    def test_one(self):
        raise AssertionError('this will fail')
```

**Parameters**

- **skip_message** – error message when test is skipped
- **exception_cls** – Exception type or tuple of Exception type that are handled to skip a test.

funq.testcase.**parameterized**(*func_suffix*, **args*, ***kwargs*)

A decorator that can generate methods given a base method and some data.

**func_suffix** is used as a suffix for the new created method and must be unique given a base method. if **func_suffix** countains characters that are not allowed in normal python function name, these characters will be replaced with "_".

This decorator can be used more than once on a single base method. The class must have a metaclass of MetaParameterized.

Example:

```
# This example will generate two methods:
#
# - MyTestCase.test_it_1
# - MyTestCase.test_it_2
#
```

```
class MyTestCase(FunqTestCase):
    __app_config_name__ = 'ma_conf'

    @parameterized("1", 5, named='nom')
    @parameterized("2", 6, named='nom2')
    def test_it(self, value, named=None):
        print value, named
```

**Parameters**

- **func_suffix** – will be used as a suffix for the new method
- ***args** – arguments to pass to the new method
- ****kwargs** – named arguments to pass to the new method

funq.testcase.**with_parameters**(*parameters*)

A decorator that can generate methods given a base method and some data. Acts like `parameterized()`, but define all methods in one call.

Example:

```
# This example will generate two methods:
#
# - MyTestCase.test_it_1
# - MyTestCase.test_it_2
#

DATA = [("1", [5], {'named':'nom'}), ("2", [6], {'named':'nom2'})]

class MyTestCase(FunqTestCase):
    app_config_name = 'ma_conf'

    @with_parameters(DATA)
    def test_it(self, value, named=None):
        print value, named
```

**Parameters** **parameters** – list of tuples (**func_suffix**, **args**, **kwargs**) defining parameters like in `todo()`.

**class** funq.testcase.**MetaParameterized**

A metaclass that allow a class to use decorators like `parameterized()` or `with_parameters()` to generate new methods.

**exception** funq.testcase.**AssertionSuccessError**(*name*)

Exception which will be raised if method decorated with `todo()` pass (it is not expected).

**Parameters** **name** – error message.

## 10.2 Configuration of tested applications

**class** funq.client.**ApplicationConfig**(*executable*, *args=()*, *funq_port=None*, *cwd=None*, *env=None*, *timeout_connection=10*, *aliases=None*, *executable_stdout=None*, *executable_stderr=None*, *attach=True*, *screenshot_on_error=False*, *with_valgrind=False*, *valgrind_args=('–leak-check=full', '–show-reachable=yes')*, *global_options=None*)

This object hold the configuration of the application to test, mostly retrieved from the funq configuration file.

Each parameter is accessible on the instance, allowing to retrieve the tested application path for example with *config.executable*, or its exeution path with *config.cwd*.

**Parameters**

- **executable** – complete path to the tested application
- **args** – executable arguments
- **funq_port** – socket port number for the libFunq connection
- **cwd** – execution path for the tested application. If None, the value will be the directory of executable.
- **env** – dict environment variables. If None, os.environ will be used.
- **timeout_connection** – timeout to try to connect to libFunq.
- **aliases** – path to the aliases file
- **executable_stdout** – file path to redirect stdout or None.
- **executable_stderr** – file path to redirect stderr or None.
- **attach** – Indicate if the process is attached or if it is a distant connection.
- **screenshot_on_error** – Indicate if screenshots must be taken on errors.
- **with_valgrind** – indicate if valgrind must be used.
- **valgrind_args** – valgrind arguments
- **global_options** – options from the funq nose plugin.

## 10.3 Entry point to communicate with a libFunq server : FunqClient

A `FunqClient` instance is generally retrieved with `funq.testcase.FunqTestcase.funq` or `funq.testcase.MultiFunqTestcase.funq`.

Example:

```
from funq.testcase import FunqTestCase

class MyTestCase(FunqTestCase):
    __app_config_name__ = 'my_conf'

    def test_something(self):
        """Method documentation"""

        my_widget = self.funq.widget('mon_widget')
```

```
        my_widget.click()

        self.funq.take_screenshot()
```

**class** funq.client.**FunqClient**(*host=None*, *port=None*, *aliases=None*, *timeout_connection=10*)
Allow to communicate with a libFunq server.

This is the main class used to manipulate tested application.

**widget**(*alias=None*, *path=None*, *timeout=10.0*, *timeout_interval=0.1*, *wait_active=True*)
Returns an instance of a funq.models.Widget or derived identified with an alias or with its complete path.

Example:

```
widget = client.widget('my_alias')
```

**Parameters**

- **alias** – alias defined in the aliases file.
- **path** – complete path for the widget
- **timeout** – if > 0, tries to get the widget until timeout is reached (second)
- **timeout_interval** – time between two atempts to get a widget (seconds)
- **wait_active** – If true - the default -, wait until the widget become visible and enabled.

**active_widget**(*widget_type='window'*, *timeout=10.0*, *timeout_interval=0.1*, *wait_active=True*)
Returns an instance of a funq.models.Widget or derived that is the active widget of the application, or the widget that got the focus.

Be careful, this method acts weidly under Xvfb.

Example:

```
my_dialog = client.active_window('modal')
```

**Parameters**

- **widget_type** – kind of widget. ('window', 'modal', 'popup' ou 'focus' -> see the QT documentation about QApplication::activeWindow, QApplication::activeModalWidget, QApplication::activePopupWidget or QApplication::focusWidget respectively)
- **timeout** – if > 0, tries to get the widget until timeout is reached (second)
- **timeout_interval** – time between two atempts to get a widget (seconds)
- **wait_active** – If true - the default -, wait until the widget become visible and enabled.

**widgets_list**(*with_properties=False*)
Returns a dict with every widgets in the application.

**dump_widgets_list**(*stream='widgets_list.json'*, *with_properties=False*)
Write in a file the result of widgets_list().

**take_screenshot**(*stream='screenshot.png'*, *format_='PNG'*)
Take a screenshot of the active desktop.

**keyclick**(*text*)
Simulate keyboard entry by sending keypress and keyrelease events for each character of the given text.

**shortcut**(*key_sequence*)
Send a shortcut defined with a text sequence. The format of this text sequence is defined with QKeySequence::fromString (see QT documentation for more details).

Example:

```
client.shortcut('F2')
```

**drag_n_drop**(*src_widget*, *src_pos=None*, *dest_widget=None*, *dest_pos=None*)
Do a drag and drop.

**Parameters**

- **src_widget** – source widget
- **src_pos** – starting position for the drag. If None, the center of *src_widget* will be used, else it must be a tuple (x, y) in widget coordinates.
- **dest_widget** – destination widget. If None, src_widget will be used.
- **dest_pos** – ending position for the drop. If None, the center of *dest_widget* will be used, else it must be a tuple (x, y) in widget coordinates.

**duplicate**()
Allow to manipulate the application in another thread.

Returns a new instance of `FunqClient` with a new socket.

Example:

```
# 'client_copy' may be used in concurrence with 'client'.
client_copy = client.duplicate()
```

## 10.4 Widgets and other classes to interact with tested application

### 10.4.1 The Widget base class

A Widget is often obtained with `funq.client.FunqClient.widget()` .

Example:

```
my_widget = self.funq.widget('my_widget')
```

**class** funq.models.**Widget**
Allow to manipulate a QWidget or derived.

**Variables**

- **client** – client for the communication with libFunq [type: `funq.client.FunqClient`]
- **oid** – ID of the managed C++ instance. [type: long]
- **path** – complete path to the widget [type: str]
- **classes** – list of class names of the managed C++ instance, in inheritance order (ie 'QObject' is last) [type : list(str)]

**properties**()
Returns a dict of availables properties for this widget with associated values.

Example:

```
enabled = widget.properties()["enabled"]
```

**set_properties**(***properties*)
Define some properties on this widget.

Example:

```
widget.set_properties(text="My beautiful text")
```

**set_property**(*name*, *value*)
Define one property on this widget.

Example:

```
widget.set_property('text', "My beautiful text")
```

**wait_for_properties**(*props*, *timeout=10.0*, *timeout_interval=0.1*)
Wait for the properties to have the given values.

Example:

```
self.wait_for_properties({'enabled': True, 'visible': True})
```

**click**(*wait_for_enabled=10.0*)
Click on the widget. If wait_for_enabled is > 0 (default), it will wait until the widget become active (enabled and visible) before sending click.

**dclick**(*wait_for_enabled=10.0*)
Double click on the widget. If wait_for_enabled is > 0 (default), it will wait until the widget become active (enabled and visible) before sending click.

**keyclick**(*text*)
Simulate keypress and keyrelease events for every character in the given text. Example:

```
widget.keyclick("my text")
```

**shortcut**(*key_sequence*)
Send a shortcut on the widget, defined with a text sequence. See the QKeySequence::fromString to see the documentation of the format needed for the text sequence.

**Parameters** **text** – text sequence of the shortcut (see QKeySequence::fromString documentation)

**drag_n_drop**(*src_pos=None*, *dest_widget=None*, *dest_pos=None*)
Do a drag and drop from this widget.

**Parameters**

- **src_pos** – starting position of the drag. Must be a tuple (x, y) in widget coordinates or None (the center of the widget will then be used)
- **dest_widget** – destination widget. If None, src_widget will be used.
- **dest_pos** – ending position (the drop). Must be a tuple (x, y) in widget coordinates or None (the center of the dest widget will then be used)

**close**()
Ask to close a widget, using QWidget::close().

**call_slot**(*slot_name*, *params={}*)
**CAUTION**; This methods allows to call a slot (written on the tested application). The slot must take a QVariant and returns a QVariant.

This is not really recommended to use this method, as it will trigger code in the tested application in an unusual way.

The methods returns what the slot returned, decoded as python object.

**Parameters**

- **slot_name** – name of the slot
- **params** – parameters (must be json serialisable) that will be send to the tested application as a QVariant.

### 10.4.2 Interacting with the data of QT Model/View framework

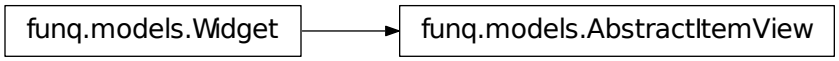
To interact with items in QAbstractTableModel, it is needed to get the associated view (QAbstractItemView). The returned instance will be of type `AbstractItemView` and the data will then be retrievable with the `AbstractItemView.model_items()` method.

Example:

```
view = self.funq.widget('my_tableview')
assert isinstance(view, AbstractItemView)

model_items = view.model_items()
item = model_items.item_by_named_path(['item1'])

item.dclick()
```




**class** funq.models.**AbstractItemView**

Specific Widget to manipulate QAbstractItemView or derived.

**model_items**()

Returns an instance of `ModelItems` based on the model associated to the view.

**current_editor**(*editor_class_name=None*)

Returns the editor actually opened on this view. One item must be in editing mode, by using `ModelItem.dclick()` or `ModelItem.edit()` for example.

Currently these editor types are handled: 'QLineEdit', 'QComboBox', 'QSpinBox' and 'QDoubleSpinBox'.

**Parameters** **editor_class_name** – name of the editor type. If None, every type of editor will be tested (this may actually be very slow)

**class** funq.models.**ModelItems**

Allow to manipulate all modelitems in a QAbstractModelItem or derived.

**Variables** **items** – list of `ModelItem`

**iter**()

Allow to iterate on every items recursively.

Example:

```
for item in items.iter():
    print item
```

**item_by_named_path**(*named_path*, *match_column=0*, *sep='/'*, *column=0*)
Returns the item (ModelItem) that match the arborescence defined by *named_path* and in the given column.

**Note:** The arguments are the same as for row_by_named_path(), with the addition of *column*.

**Parameters** **column** – the column of the desired item

**row_by_named_path**(*named_path*, *match_column=0*, *sep='/'*)
Returns the item list of ModelItem that match the arborescence defined by *named_path*, or None if the path does not exists.

**Important:** Use unicode characters in *named_path* to match elements with non-ascii characters.

Example:

```
model_items.row_by_named_path([u'TG/CBO/AP (AUT 1)',
                               u'Paramètres tranche',
                               u'TG',
                               u'DANGER'])
```

**Parameters**

- **named_path** – path for the interesting ModelIndex. May be defined with a list of str or with a single str that will be splitted on *sep*.
- **match_column** – column used to check‘named_path‘ is a string.

**class** funq.models.**ModelItem**
Allow to manipulate a modelitem in a QAbstractModelItem or derived.

**Variables**

- **viewid** – ID of the view attached to the model containing this item [type: long]
- **row** – item row number [type: int]
- **column** – item column number [type: int]
- **value** – item text value [type: unicode]
- **check_state** – item text value of the check state, or None
- **itempath** – Internal ID to localize this item [type: str ou None]
- **items** – list of subitems [type: ModelItem]

**select**()
Select this item.

**edit**()
Edit this item.

**click**()
Click on this item.

**dclick**()
Double click on this item.

**is_checkable**()
Returns True if the item is checkable

**is_checked**()
Returns True if the item is checked

### 10.4.3 Interacting with the data of QT Graphics View framework

Handling QGraphicsItems data is quite similar to handling data of the Models/Views framework.

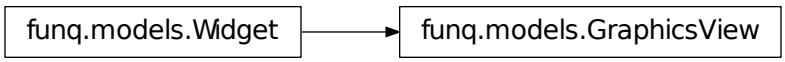
It requires the associated view (an instance of QGraphicsView). In funq the widget will be an instance of `GraphicsView` and the data will be available with the `GraphicsView.gitems()` method.

Example:

```
gview = self.funq.widget('my_gview')

gitems = gview.gitems()

for item in gitems.iter():
    # do something with item
```




**class** funq.models.**GraphicsView**
Allow to manipulate an instance of QGraphicsView.

**gitems**()
Returns an instance of `GItems`, that will contains every items of this QGraphicsView.

**dump_gitems**(*stream='gitems.json'*)
Write in a file the list of graphics items.

**class** funq.models.**GItems**
Allow to manipulate a group of QGraphicsItems.

**Variables** **items** – list of `GItem` that are on top of the scene (and not subitems)

**iter**()
Allow to iterate on every items recursively.

Example:

```
for item in items.iter():
    print item
```

**class** funq.models.**GItem**
Allow to manipulate a QGraphicsItem.

**Variables**

- **viewid** – ID of the view attached to the model containing this item [type: long]
- **stackpath** – Internal gitem ID, based on stackIndex and parent items [type: str]
- **objectname** – value of the "objectName" property if it inherits from QObject. [type: unicode or None]
- **classes** – list of names of class inheritance if it inherits from QObject. [type: list(str) or None]
- **items** – list of subitems [type: GItem]

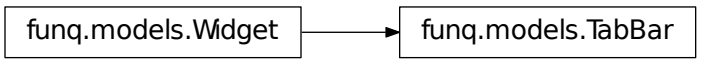
**is_qobject**()
Returns True if this GItem inherits QObject

**properties**()
Return the properties of the GItem. The GItem must inherits from QObject.

**click**()
Click on this gitem.

**dclick**()
Double click on this gitem.

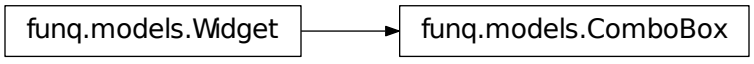
### 10.4.4 Other widgets




**class** funq.models.**TabBar**
Allow to manipulate a QTabBar Widget.

**tab_texts**()
Returns the list of texts in tabbar.

**set_current_tab**(*tab_index_or_name*)
Define the current tab given an index or a tab text.




**class** funq.models.**ComboBox**
Allow to manipulate a QCombobox.

**model_items**()
Returns the items (ModelItems) associated to this combobox.

**set_current_text**(*text*)
Define the text of the combobox, ensuring that it is a possible value.

CHAPTER 11

# Indices and references

- *genindex*
- *modindex*
- *search*