# The Instrumentino software platform, Version 1 (04/2014) Instruction Manual

Written By:

- Joel Koenka

- Roland Fuiko

## Description

*Instrumentino* is a graphical user interface (GUI) program, written in Python (version 2.7), which allows control of experimental settings, using controllers such as Arduino. It also includes a lean software package for the Arduino itself for the communication protocol (*Controlino*).

## Motivation

The motivation to develop this program was to create an easy-to-use powerful way to control purpose made systems. Programming is not something that most people are comfortable with, so *Instrumentino* is built to minimize the programming efforts required, while giving a comfortable uniform way to control and run experiments. Python is well suited for that purpose, being a very popular high-end language, which is easily learned and for which plenty of online help exists in forums (just google – “how to do this and that using Python”).

## Features

A single system dependent file describes system components (e.g. valves, pumps, pressure and voltage controllers) with their relevant system variables (e.g. valve state, plunger location, pressure, voltage and current) and basic actions, which change these variables in order to achieve something (e.g. flush interface, flush capillary, perform a HV separation); each action can receive parameters (such as pressure, time, voltage).

These actions can be then added in a list to create methods (can be saved as a *.mtd* files and loaded later), and saved methods can be added to a list to create sequences (can be saved as *.seq* files and loaded as well). A method or a sequence can be run by the user and stopped when needed, using the big STOP button. During each run, each action is logged to a file for future reference.

After initial connection to the relevant hardware controller (such as an Arduino), system variables are monitored and their values are periodically updated on the left panel, as well as graphically plotted in the log panel.

## Requirements

In order to use *Instrumentino*, you should know Python to some extent and install some programs in order to create and run the project.

A short list of necessary steps:

* Download and untar the *instrumentino* package from the CPC Program Library.
* Install Python 2.7, 32 bit (e.g. from <https://www.python.org/download/releases/2.7>)
* Install (using *easy\_install*) the *Instrumentino* egg file (*supplementary material/Instrumentino/dist)*.
* Install the following Python packages:
  + pySerial
  + wxPython (“agw” might be necessary as well)
  + matplotlib
* Create a system description file (e.g. *supplementary material/example run (container.py)/container.py*) See section 5.
* Execute the file – this will start the software

There are several ways of installing the necessary Python software, the one I prefer is *Enthought Canopy*. *Canopy* is a Python distribution that includes (among other things) a visual way to install python packages.

In order to edit and run python programs, we recommend the open source *Eclipse* integrated development environment (IDE).

## Install on Mac OS

Following are the steps for starting to use *Instrumentino* on a Mac, using *Canopy* and *Eclipse*.

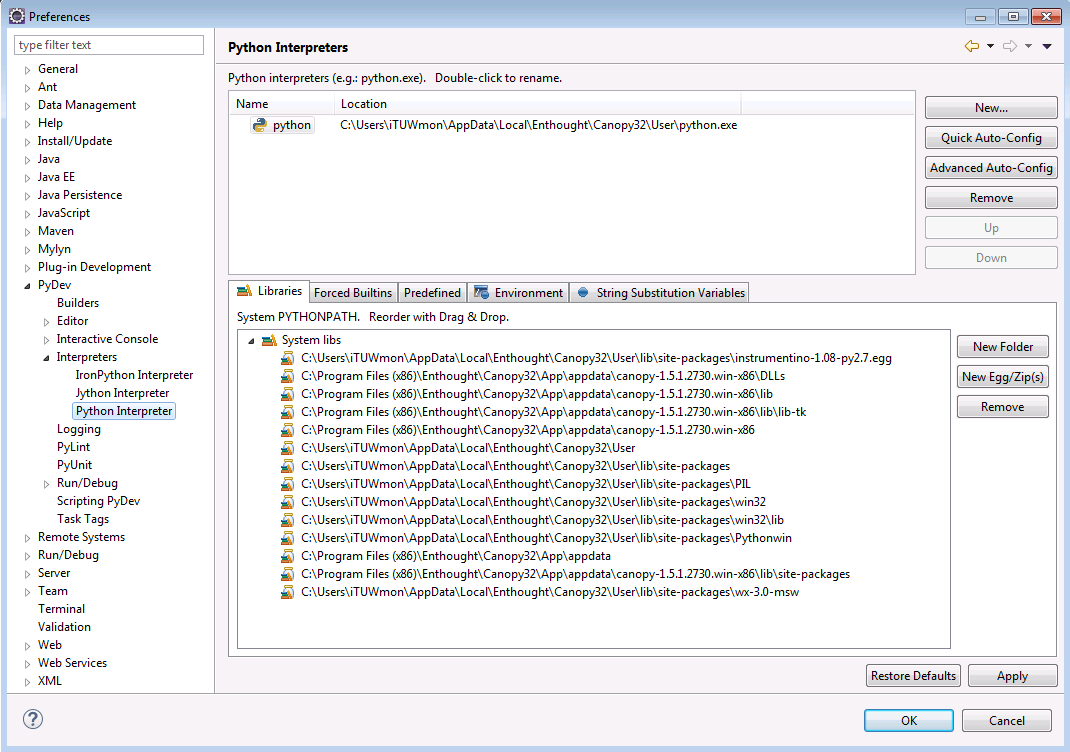
* Install *Canopy*:  
  <https://www.enthought.com/downloads/>  
  This will install Python and some common packages in the computer. You need to register first with an academic email address and request an academic license (this takes only a few minutes to complete). Only then can you download the *Canopy Full Installer*, please use the 32 bit version.  
  To check if the installation was done properly, open a Terminal and type python. Check that the Python version is 2.7.X, as *Instrumentino* was written using python 2.7
* Open *Canopy*, and choose “Package manager”. Install the following packages: wxPython, matplotlib, pyserial
* To write and test your code, an IDE (Integrated Development Environment) would come in handy. A good free option is *Eclipse*:  
  <http://www.eclipse.org/downloads/>  
  Install either the *Standard* version or the *Java* version of the newest stable release. If it gives you error (from unidentified user) then go to system preferences and Tick window, which allows you to open *Eclipse*.  
  *Eclipse* is an IDE that supports many programming languages, and the program became so big that the developers decided to release different versions of *Eclipse*. In order to develop Python programs in *Eclipse*, an additional plugin named PyDev (described next) is needed. From my experience, this works for *Eclipse* versions supporting Java but not for *Eclipse* versions for C/C++.
* Open *Eclipse*, and install PyDev. To install plugins in *Eclipse*:
  + Go to the Help menu and choose *Install New Software*.
  + Click the *Add* button and enter these details:
    - Name: PyDev (This doesn’t really matter)
    - Location: <http://pydev.org/updates>
  + Uncheck the *'Contact all update sites during install to find required software’*, as
  + Tick everything in the window and continue according to the instructions. Agree when asked to trust the certificates.
  + Restart *Eclipse*
* Install the *Instrumentino* framework egg by running the following command in the directory where it is stored (*supplementary material/Instrumentino/dist*):  
  easy\_install instrumentino-1.0-py2.7.egg (replace the filename if needed).  
  Test if it’s installed by running python in the terminal and typing import instrumentino
* Create a *workspace* directory for code projects e.g. *Documents/workspace/Instruments/mySystem* might be a good choice.
* Start a PyDev project. It should have one python file for the system description (e.g. *mySystem.py*). For an easy start, use a ready file from another system (e.g. *supplementary material/example run (container.py)/container.py*) and change it, according to your system’s requirements.
* To run the code, open the file *mySystem.py* and while in PyDev view, choose the menu *Run>Run*. Alternatively, you can open a terminal, navigate to where *mySystem.py* is and type python mySystem.py

## Install on Windows 7

Following are the steps for starting to use *Instrumentino* on Windows 7 (32bit or 64bit), using *Canopy* and *Eclipse*.

* Install *Canopy 1.5.1*:  
  <https://www.enthought.com/downloads/>  
  This will install Python and some common packages in the computer. You need to register first with an academic email address and request an academic license (this takes only a few minutes to complete). Only then can you download the *Canopy Full Installer*, please use the 32 bit version.  
  To check if the installation was done properly, open a Terminal (cmd.exe) and type C:\python. Check that the Python version is 2.7.X, as *Instrumentino* was written using python 2.7
* Open Canopy 1.5.1 and uninstall the wxPython 2.8.10.1-7 package with the “Package Manager”. Further check the versions of the following packages:
  + Matplotlib 1.4.2-2
  + Pyserial 2.6-1

If the versions differ, the appropriate versions can be installed via the package manager, or downloaded at the according website.

* Download and install the wxPython 3.0.2 package for 32bit and Python 2.7  
  <http://downloads.sourceforge.net/wxpython/wxPython3.0-win32-3.0.2.0-py27.exe>
* To write and test your code, an IDE (Integrated Development Environment) would come in handy. A good free option is *Eclipse*:  
  <http://www.eclipse.org/downloads/>  
  Install the Eclipse IDE for Java EE Developers (LUNA 4.4.1)   
  *Eclipse* is an IDE that supports many programming languages, and the program became so big that the developers decided to release different versions of *Eclipse*. In order to develop Python programs in *Eclipse*, an additional plugin named PyDev (described next) is needed.
* Open *Eclipse*, and install PyDev. To install plugins in *Eclipse*:
  + Go to the Help menu and choose *Install New Software*.
  + Click the *Add* button and enter these details:
    - Name: PyDev (This doesn’t really matter)
    - Location: <http://pydev.org/updates>
  + Uncheck the *'Contact all update sites during install to find required software’*, as
  + Tick everything in the window and continue according to the instructions. Agree when asked to trust the certificates.
  + Restart *Eclipse*
* A big advantage of Eclipse is the handling of the PYTHONPATH environment. If any errors occur during the install process of the packages mentioned above, start eclipse and navigate to: Window → Preference → Pydev → Interpreter Python menu. Here you have the possibility to change the path to the python interpreter and the installed packages. Usually the Eclipse IDE handle that for your needs. The result should look like the following:   
  
* Download *Instrumentino* <https://github.com/yoelk/Instrumentino>and unzip it.
* Install the *Instrumentino* framework egg by running the following command in the directory where it is stored (\Instrumentino-master\instrumentino\dist):
  + Start a command shell (cmd.exe)
  + Type in: C:\easy\_install instrumentino-1.08-py2.7.egg (replace the filename if needed).
  + Test if it’s installed by running c:\python in the terminal and typing:  
    >>> import instrumentino
* Create a *workspace* directory for code projects e.g. *Documents/workspace/Instruments/mySystem* might be a good choice.
* Start Eclipse and a PyDev project. It should have one python file for the system description (e.g. *mySystem.py*). For an easy start, use a ready file from another system (e.g. *supplementary material/example run (container.py)/container.py*) and change it, according to your system’s requirements.
* To run the code, open the file *mySystem.py* and while in PyDev view, choose the menu *Run>Run*. Alternatively, you can open a terminal, navigate to where *mySystem.py* is and type python mySystem.py

**Troubleshooting and install hints:**

* >>> import instrumentino throws an error containing “UltimateListCtrl” or something with “wxversion” then something is wrong with the wxPython installation. Try to remove every installed package of wxPython and run a registry cleaner. After that start Eclipse and navigate to: Window → Preference → Pydev → Interpreter Python menu. Check for any wxPython paths. If everything is clean, install the wxPython 3.0.2 package for 32bit and Python 2.7.   
  When the error wxversion occurs usually two versions of wxPython are installed. The “UltimateListCtrl” error comes with the wrong version of wxPython.  
  This problem may be also solved by using the Canopy package manager to install the package “agw” (version 0.9.1 is ok).
* Canopy Package Manager gives the error that no internet connection is available: Solution: logging out from and logging again in to your Canopy account. You can only be logged in on one machine!!
* C:\easy\_install instrumentino-1.08-py2.7.egg went wrong:  
  Open Eclipse and navigate to: Window → Preference → Pydev → Interpreter Python menu. Remove all entries from PYTHONPATH environment variable and remove the Python interpreter (python27.exe). Add the interpreter again. The PYTHONPATH variables will be generated automatically. -> Try again.

## The system description file

The system description file (e.g. *mySystem.py*) should include the following parts:

* **imports**As in any Python file, it should begin with imports of other Python files it refers to (similar to *include* statements in C++).  
  It should at least include these imports:
  + from \_\_future\_\_ import division
  + from instrumentino import Instrument
  + from instrumentino import cfg
  + from instrumentino.action import SysAction

Import the necessary controller classes for your system. For example:

* + from instrumentino.controllers.arduino.parker import ParkerPressureController
  + from instrumentino.controllers.labsmith\_eib.labsmith\_comps import LabSmithValves4VM01
  + from instrumentino.controllers.labsmith\_eib import SysVarDigitalLabSmith\_AV201Position
  + Then, import the necessary parameters for the actions you’ll define later. For example:
  + from instrumentino.action import SysActionParamTime, SysActionParamFloat
* **Systems constants**This may include Arduino pin assignments (e.g. pinAnalInParkerP = 5) or any other number which needs to be used in the rest of the code. It’s always good practice while coding to give meaningful names to numbers, so the code becomes readable and understandable.
* **System components**This section describes the components in the system. Each component is represented by a Python class from the *Instrumentino* package. To check what hardware components are supported, browse the Instrumentino code and look in the path *instrumentino/controllers/.* There are some components already defined there, but if yours isn’t, you need to create it; this will not be covered in this manual.  
  As an example, here is a definition line for a Parker pressure controller:  
  pressureController = ParkerPressureController('Pressure', (0,100), pinAnalInParkerP, pinPwmOutParkerP, highFreqPWM=True)  
  The first given argument is the display name on the screen. The second is the allowed pressure range. The third and fourth are the input and output Arduino pins, used to control it. The fifth modifies the PWM frequency used for the analog output signal.   
  The order of these parameters can be checked in the *Instrumentino* code at *instrumentino/controllers/arduino.parker.py*
* **System actions**This section lists the possible basic actions to be defined for the system. Here is an example for an action fills a container, using a pressure controller (see *Instumentino* release article):  
  class SysActionFillContainer(SysAction):

def \_\_init\_\_(self):

self.seconds = SysActionParamTime()

self.psi = SysActionParamFloat(pressureController.vars['P'])

SysAction.\_\_init\_\_(self, 'Fill Container', (self.seconds, self.psi))

def Command(self):

# Connect container to pressure controller

valves.vars['V1'].Set(valvePortPressure)

# Start pressure

pressureController.vars['P'].Set(self.psi.Get())

# Wait some time

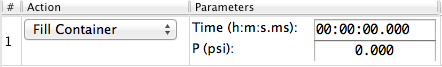
cfg.Sleep(self.seconds.Get())

# Close container

valves.vars['V1'].Set('closed')

The action is defined as a class which inherits from *SysAction* (imported in the import section), and implements two necessary functions: *\_\_init\_\_(self)* and *Command(self).*   
*\_\_init\_\_(self)* is an initialization function, in which the required parameters for this action are defined. In this example: a time parameter and a pressure parameter.

These parameters are then passed on to the parent class’s *\_\_init\_\_* function, as arguments. Afterwards, when using the software, this is how it looks:



* **System definition**In this section, the former components and actions are wrapped up by a single class called *System*, which inherits from *instrumentino.Instrument*.  
  Here is an example:class System(Instrument):

def \_\_init\_\_(self):

comps = (pressureController, valves)

actions = (SysActionFillContainer(),

SysActionEmptyContainer())

name = 'Example System'

description = 'A container connected to a pressure controller'

version = '1.0'

Instrument.\_\_init\_\_(self, comps, actions, version, name, description)  
As can be seen, the various components are given in a tuple to the variable *comps*, the actions to the variable *actions*, and the name, description and version string are defined as well. They’re all given to the *\_\_init\_\_* function of *instrumentino.Instrument* in the appropriate order. Once this is defined, the *Instrumentino* framework will use this class to create the appropriate control program.

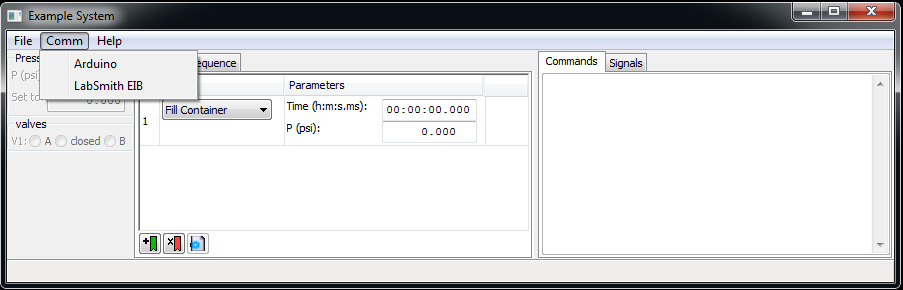
* **Program run**This section shows how to make the python file runnable. The only thing needed is to initiate a *System* class, as defined in the last section. This starts the program.  
  if \_\_name\_\_ == '\_\_main\_\_':

# run the program

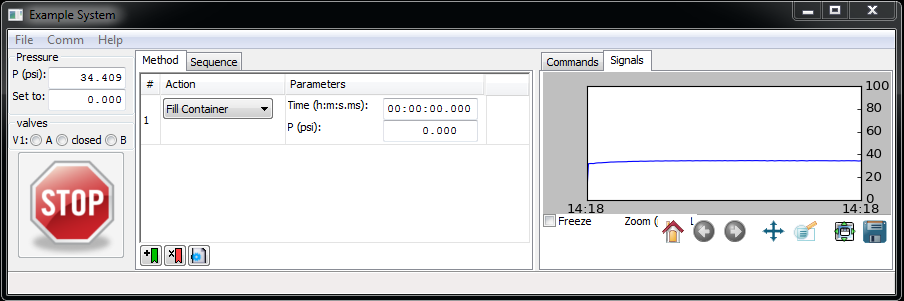
System()

## Using the program

The program opens and waits for the relevant controllers to be connected through a serial port. At the moment, there is support for an Arduino (running the *Controlino* code) and a LabSmith EIB. Use the *Comm* menu to get the system online.



When the system is online, the system components’ pane is updated with values for the system variables and the signal log can be seen as well. Now it’s possible to run methods or scripts



Use the *File* menu to save/load methods and sequences.

## Preparation of a stand-alone executable file (main.exe)

For extra reading about this process:

* <http://www.pyinstaller.org/>
* <http://stackoverflow.com/questions/7674790/bundling-data-files-with-pyinstaller-onefile>
* [http://irwinkwan.com/2013/04/29/Python-executables-pyinstaller-and-a-48-hour-game-design-compo/](http://irwinkwan.com/2013/04/29/python-executables-pyinstaller-and-a-48-hour-game-design-compo/)

For Windows:

In order to make a stand-alone executable, to be transferred to another windows PC (which doesn’t have Python installed on it), it’s necessary to download PyInstaller:

<http://www.pyinstaller.org/>.

Download it to your PC, let’s say to C:\programs\pyinstaller-2.0\

Then, while in the directory of your system description file, type in the terminal:  
C:\programs\pyinstaller-2.0\pyinstaller.py mySystem.py  
This creates a new directory *Instrumentino\dist\mySystem*, in which there’s an executable file to run the program. Because of a bug in PyInstaller, it is necessary to copy the *Instrumentino* code to this directory.