RobotPy Documentation

Release 2017

RobotPy development team

January 18, 2017

Welcome! RobotPy is a project created by a community of FIRST mentors and students dedicated to developing python-related projects for the FIRST Robotics Competition. This documentation site contains information about various projects that RobotPy supports, including guides and API references.

Note: RobotPy is a community project and the tools we create are not officially supported by FIRST. Please see the FAQ for more information.

CHAPTER 1

Projects

The primary reason RobotPy exists is to support teams that want to write their FRC robot code using Python, and we have several projects related to this:

- robotpy-wpilib: the python implementation of WPILib for FRC
- pyfrc: provides unit testing, realtime robot simulation, and easy upload capabilities for your RobotPy code
- roborio-packages: Various python packages for the RoboRIO platform installable by opkg, including the python interpreter and numpy
- robotpy-wpilib-utilities: Community focused extensions for WPILib

Additionally, RobotPy is home to several projects that are useful for all teams, even if they aren't writing their robot code in python:

- pynetworktables: python bindings for NetworkTables that you can use to communcate with SmartDashboard and/or your robot.
- pynetworktables2js: Forwards NetworkTables traffic to a web page, allowing you to write custom dashboards for your robot using HTML/Javascript
- pynetconsole: A simple netconsole implementation in python
- roborio-vm: Scripts to create a QEMU virtual machine from the RoboRIO image file

There is a lot of good documentation, but there's still room for improvement. We welcome contributions from all members of the FIRST community!

1.1 Getting Started

RobotPy WPILib is a set of libraries that are used on your roboRIO to enable you to use Python as your main programming language for FIRST Robotics robot development. It includes support for almost all components that are supported by WPILib's Java implementation.

You can run RobotPy-based programs either on your computer or on a robot. There are a lot of different ways you can get started with RobotPy, but we recommend the following steps:

- *Install the Robot Simulator and related tools*
- Learn how to write Python-based robot code via *Anatomy of a robot* and the various sections of the *programmer's guide*

Once you've played around with some code in simulation, then you should *install RobotPy on your robot*.

If you're looking to use pynetworktables on the driver station or on a coprocessor, then check out the *pynetworktables install docs*.

1.2 Installation

- To install RobotPy on your robot, see the *RobotPy installation documentation*.
- If you wish to deploy code or use the Robot Simulator, see the *pyfrc installation documentation*.
- To install pynetworktables on a system that does not have RobotPy or pyfrc installed on it (such as a coprocessor), see the *pynetworktables installation documentation*.

1.2.1 Robot Installation

Note: If you install RobotPy on your RoboRIO, you are still able to deploy C++ and Java programs without any conflicts.

Automated installation

Warning: This guide assumes that your RoboRIO has the current legal RoboRIO image installed. If you haven't done this yet, see the screensteps documentation for imaging instructions. To image the RoboRIO for RobotPy, you only need to have the latest FRC Update Suite installed.

RobotPy is truly cross platform, and can be installed from Windows, most Linux distributions, and from Mac OSX also. Here's how you do it:

- Download RobotPy from github
- Make sure Python 3.4 or newer is installed

Unzip the RobotPy zipfile somewhere on your computer (not on the roboRIO), and there should be an installer.py there. Open up a command line, change directory to the installer location, and run this:

```
Windows:   py -3 installer.py install-robotpy

Linux/OSX: python3 installer.py install-robotpy
```

It will ask you a few questions, and copy the right files over to your robot and set things up for you.

Next, you'll want to create some code (or maybe use one of our examples), and upload it to your robot! Refer to our *Programmer's Guide* for more information.

Upgrading

From the same directory that you unzipped previously, you can run the same installer script to upgrade your robotpy installation. You need to do it in two phases, one while connected to the internet to download the new release, and one while connected to the Robot's network.

When connected to the internet:

```
Windows:   py installer.py download-robotpy

Linux/OSX: python3 installer.py download-robotpy
```

Then connect to the Robot's network:

```
Windows:   py installer.py install-robotpy

Linux/OSX: python3 installer.py install-robotpy
```

If you want to use a beta version of RobotPy (if available), you can add the –pre argument to the download/install command listed above.

Manual installation (release)

Warning: This isn't recommended, so you're on your own if you go this route.

If you really want to do this, it's not so bad, but then you lose out on the benefits of the automated installer – in particular, this method requires internet access to install the files on the roboRIO in case you need to reimage your roboRIO.

- Connect your roboRIO to the internet
- SSH in, and copy the following to /etc/opkg/robotpy.conf:

```
src/gz robotpy http://www.tortall.net/~robotpy/feeds/2017
```

- Run this:

```
opkg install python36 netconsole-host
```

- Then run this:

```
pip3 install robotpy-hal-roborio wpilib
```

Note: When powered off, your roboRIO does not keep track of the correct date, and as a result pip may fail with an SSL related error message. To set the date, you can either:

- Set the date via the web interface
- You can login to your roboRIO via SSH, and set the date via the date command:

```
date -s "2015-01-03 00:00:00"
```

Upgrading requires you to run the same commands, but with the appropriate flags set to tell pip3/opkg to upgrade the packages for you.

1.2.2 pyfrc install

Installing pyfrc will install all of the packages needed to work with RobotPy on your system, including WPILib, pynetworktables, unit testing support, and the *robot simulator*.

pyfrc requires Python 3.4 or greater to be installed on your computer.

- Python for Windows

- Python for OSX

Install via pip on Windows

Note: pip typically requires internet access

The easiest installation is by using pip. pip is installed by default with Python 3.4. Run the following command from the command line:

```
py -3 -m pip install pyfrc
```

To upgrade, you can run this:

```
py -3 -m pip install --upgrade pyfrc
```

If you don't have administrative rights on your computer, either use virtualenv/virtualenvwrapper-win, or or you can install to the user site-packages directory:

```
py -3 -m pip install --user pyfrc
```

Install via pip on OSX/Linux

Note: pip typically requires internet access

The easiest installation is by using pip. pip is installed by default with Python 3.4 (though on Linux it may not be installed). On a Linux or OSX system that has pip installed, just run the following command from the Terminal applciation (may require admin rights):

```
pip3 install pyfrc
```

To upgrade, you can run this:

```
pip3 install --upgrade pyfrc
```

If you don't have administrative rights on your computer, either use virtualenv/virtualenvwrapper, or or you can install to the user site-packages directory:

```
pip3 install --user pyfrc
```

Manual install (without pip)

While this is possible to do, due to the large number of dependencies this is not recommended nor is it supported.

code coverage support

If you wish to run code coverage testing, then you must install the coverage package. It requires a compiler to install from source. However, if you are using a supported version of Python and a modern version of pip, it may install a binary wheel instead, which removes the need for a compiler.

```
Windows:   py -3 -m pip install coverage

Linux/OSX: pip3 install coverage
```

If you run into compile errors, then you will need to install a compiler on your system.

- On Windows you can download the Visual Studio compilers for Python (be sure to download the one for your version of Python).
- On OSX it requires XCode to be installed
- On Linux you will need to have python3-dev/python3-devel or a similar package installed

1.2.3 pynetworktables install

pynetworktables requires Python 2.7 or 3.3 or greater to be installed on the system that you'll be using it on.

Note: You only need to install pynetworktables separately if you're using it on a system that doesn't already have pyfrc or RobotPy installed on it (such as a coprocessor)

Install via pip on Windows

The latest versions of Python on Windows come with pip, but you may need to install it by hand if you're using an older version. Once pip is installed, run the following command from the command line:

```
Python 2.7: py -2 -m pip install pynetworktables
Python 3.x: py -3 -m pip install pynetworktables
```

To upgrade, you can run this:

```
Python 2.7: py -2 -m pip install --upgrade pynetworktables
Python 3.x: py -3 -m pip install --upgrade pynetworktables
```

If you don't have administrative rights on your computer, either use virtualenv/virtualenvwrapper-win, or or you can install to the user site-packages directory:

```
Python 2.7: py -2 -m pip install --user pynetworktables
Python 3.x: py -3 -m pip install --user pynetworktables
```

Install via pip on OSX/Linux

On a Linux or OSX system that has pip installed, just run the following command from the Terminal application (may require admin rights):

```
Python 2.7: pip install pynetworktables
Python 3.x: pip3 install pynetworktables
```

To upgrade, you can run this:

```
Python 2.7: pip install --upgrade pynetworktables
Python 3.x: pip3 install --upgrade pynetworktables
```

If you don't have administrative rights on your computer, either use virtualenv/virtualenvwrapper-win, or or you can install to the user site-packages directory:

```
Python 2.7: pip -m pip install --user pynetworktables
Python 3.x: pip3 -m pip install --user pynetworktables
```

Manual install (without pip)

Note: It is highly recommended to use pip for installation when possible

You can download the source code, extract it, and run this:

```
python setup.py install
```

If you are using Python 2.7, you will need to also install the monotonic package from pypi

1.3 Programmer's Guide

1.3.1 Anatomy of a robot

Note: The following assumes you have some familiarity with python, and is meant as a primer to creating robot code using the python version of wpilib. If you're not familiar with python, you might try these resources:

- List of various guides to learn Python
- CodeAcademy
- Python 3.5 Tutorial

This tutorial will go over the things necessary for very basic robot code that can run on an FRC robot using the python version of WPILib. Code that is written for RobotPy can be ran on your PC using various simulation tools that are available.

Create your Robot code

Your robot code must start within a file called `robot.py`. Your code can do anything a normal python program can, such as importing other python modules & packages. Here are the basic things you need to know to get your robot code working!

Importing necessary modules

All of the code that actually interacts with your robot's hardware is contained in a library called WPILib. This library was originally implemented in C++ and Java. Your robot code must import this library module, and create various objects that can be used to interface with the robot hardware.

To import wpilib, it's just as simple as this:

```
import wpilib
```

Note: Because RobotPy implements the same WPILib as C++/Java, you can learn a lot about how to write robot code from the many C++/Java focused WPILib resources that already exist, including FIRST's official documentation. Just translate the code into python.

Robot object

Every valid robot program must define a robot object that inherits from either `wpilib.IterativeRobot` or `wpilib.SampleRobot`. These objects define a number of functions that you need to override, which get called at various times.

- `wpilib.IterativeRobot` functions
- `wpilib.SampleRobot` functions

Note: It is recommended that inexperienced programmers use the IterativeRobot framework, which is what this guide will discuss.

An incomplete version of your robot object might look like this:

```
class MyRobot(wpilib.IterativeRobot):

    def robotInit(self):
        self.motor = wpilib.Jaguar(1)
```

The `robotInit` function is where you initialize data that needs to be initialized when your robot first starts. Examples of this data includes:

- Variables that are used in multiple functions
- Creating various wpilib objects for devices and sensors
- Creating instances of other objects for your robot

In python, the constructor for an object is the `__init__` function. Instead of defining a constructor for your main robot object, you can override robotInit instead. If you do decide that you want to override `__init__`, then you must call `super().__init__()` in your `__init__` method, or an exception will be thrown.

Adding motors and sensors

Everything that interacts with the robot hardware directly must use the wpilib library to do so. Starting in 2015, full documentation for the python version of WPILib is published online. Check out the API documentation (`wpilib`) for details on all the objects available in WPILib.

Note: You should *only* create instances of your motors and other WPILib hardware devices (Gyros, Joysticks, Sensors, etc) either during or after robotInit is called on your main robot object. If you don't, there are a lot of things that will fail.

Creating individual devices

Let's say you wanted to create an object that interacted with a Jaguar motor controller via PWM. First, you would read through the table (`wpilib`) and see that there is a `Jaguar` object. Looking further, you can see that the constructor

takes a single argument that indicates which PWM port to connect to. You could create the *Jaguar* object that is using port 4 using the following python code in your *robotInit* method:

```
self.motor = wpilib.Jaguar(4)
```

Looking through the documentation some more, you would notice that to set the PWM value of the motor, you need to call the `Jaguar.set()` function. The docs say that the value needs to be between -1.0 and 1.0, so to set the motor full speed forward you could do this:

```
self.motor.set(1)
```

Other motors and sensors have similar conventions.

Robot drivetrain control

For standard types of drivetrains (2 or 4 wheel, and mecanum), you'll want to use the `RobotDrive` class to control the motors instead of writing your own code to do it. When you create a RobotDrive object, you either specify which PWM channels to automatically create a motor for:

```
self.robot_drive = wpilib.RobotDrive(0,1)
```

Or you can pass in motor controller instances:

```
l_motor = wpilib.Talon(0)
r_motor = wpilib.Talon(1)
self.robot_drive = wpilib.RobotDrive(l_motor, r_motor)
```

Once you have one of these objects, it has various methods that you can use to control the robot via joystick, or you can specify the control inputs manually.

See also:

Documentation for the `wpilib.RobotDrive` object, and the FIRST WPILib Programming Guide.

Robot Operating Modes (IterativeRobot)

During a competition, the robot transitions into various modes depending on the state of the game. During each mode, functions on your robot class are called. The name of the function varies based on which mode the robot is in:

- `disabledXXX` - Called when robot is disabled
- `autonomousXXX` - Called when robot is in autonomous mode
- `teleopXXX` - Called when the robot is in teleoperated mode
- `testXXX` - Called when the robot is in test mode

Each mode has two functions associated with it. xxxInit is called when the robot first switches over to the mode, and xxxPeriodic is called 50 times a second (approximately – it's actually called as packets are received from the driver station).

For example, a simple robot that just drives the robot using a single joystick might have a teleopPeriodic function that looks like this:

```
def teleopPeriodic(self):
    self.robot_drive.arcadeDrive(self.stick)
```

This function gets called over and over again (about 50 times per second) while the robot remains in teleoperated mode.

Warning: When using the IterativeRobot as your Robot class, you should avoid doing the following operations in the xxxPeriodic functions or functions that have xxxPeriodic in the call stack:

- Never use `Timer.delay()`, as you will momentarily lose control of your robot during the delay, and it will not be as responsive.
- Avoid using loops, as unexpected conditions may cause you to lose control of your robot.

Main block

Languages such as Java require you to define a 'static main' function. In python, because every .py file is usable from other python programs, you need to define a code block which checks for __main__. Inside your main block, you tell WPILib to launch your robot's code using the following invocation:

```
if __name__ == '__main__':
    wpilib.run(MyRobot)
```

This simple invocation is sufficient for launching your robot code on the robot, and also provides access to various RobotPy-enabled extensions that may be available for testing your robot code, such as pyfrc and robotpy-frcsim.

Putting it all together

If you combine all the pieces above, you end up with something like this below, taken from one of the samples in our github repository:

```
#!/usr/bin/env python3
"""
    This is a good foundation to build your robot code on
"""

import wpilib

class MyRobot(wpilib.IterativeRobot):

    def robotInit(self):
        """
        This function is called upon program startup and
        should be used for any initialization code.
        """
        self.robot_drive = wpilib.RobotDrive(0,1)
        self.stick = wpilib.Joystick(1)

    def autonomousInit(self):
        """This function is run once each time the robot enters autonomous mode."""
        self.auto_loop_counter = 0

    def autonomousPeriodic(self):
        """This function is called periodically during autonomous."""

        # Check if we've completed 100 loops (approximately 2 seconds)
        if self.auto_loop_counter < 100:
            self.robot_drive.drive(-0.5, 0) # Drive forwards at half speed
            self.auto_loop_counter += 1
        else:
            self.robot_drive.drive(0, 0)    #Stop robot

    def teleopPeriodic(self):
```

```
        """This function is called periodically during operator control."""
        self.robot_drive.arcadeDrive(self.stick)

    def testPeriodic(self):
        """This function is called periodically during test mode."""
        wpilib.LiveWindow.run()

if __name__ == "__main__":
    wpilib.run(MyRobot)
```

There are a few different python-based robot samples available, and you can find them in our github examples repository.

Next Steps

This is a good foundation for building your robot, next you will probably want to know about *Running robot code*.

1.3.2 Running robot code

Now that you've created your first Python robot program, you probably want to know how to run the code. The process to run a python script is slightly different for each operating system.

Note: This section assumes that you've already *installed pyfrc*. If you haven't, now's a great time to do so!

How to execute the script

Windows

On Windows, you will typically execute your robot code by opening up the command prompt (cmd), changing directories to where your robot code is, and then running this:

```
py -3 robot.py
```

Linux/OSX

On Linux/OSX, you will typically execute your robot code by opening up the Terminal program, changing directories to where your robot code is, and then running this:

```
python3 robot.py
```

Eclipse

If you're using pydev with Eclipse, there are a couple of ways to run the code.

- Right click on the file in project explorer, and select `PyDev -> Run As -> Python Run`
- Right click on the text editor and select `Run As -> Python Run`

After running the python script the first time, you'll want to pass it arguments (see below). To edit the arguments, click on the little arrow next to the green play button, and select "Run Configurations". Under "Python Run", you can select your configuration, and then select the "arguments" tab.

Commands

When you run your code without additional arguments, you'll see an error message saying something like `robot.py: error: the following arguments are required: command`. RobotPy tools install various commands that you can run from your robot code. To discover the various features that are installed, you can use the `--help` command:

```
Windows:   py -3 robot.py --help

Linux/OSX: python3 robot.py --help
```

Note: RobotPy supports an extension mechanism that allows advanced users the ability to create their own custom `robot.py` commandline options. For more information, see *Adding options to robot.py*

Next steps

There are two ways you can run the code: on the robot, and on the simulator:

- *Deploying to the robot*
- *Robot Simulator*

1.3.3 Deploying to the robot

The easiest way to install code on the robot is to use the deploy command provided by pyfrc. This command will first run any unit tests on your robot code, and if they pass then it will upload the robot code to the roboRIO. Running the tests is really important, it allows you to catch errors in your code before you run it on the robot.

1. Make sure you have RobotPy installed on the robot (*RobotPy install guide*)
2. Make sure you have pyfrc installed (*pyfrc install guide*)
3. Once that is done, you can just run the following command and it will upload the code and start it immediately.

```
Windows:   py -3 robot.py deploy

Linux/OSX: python3 robot.py deploy
```

You can watch your robot code's output (and see any problems) by using the netconsole program (you can either use NI's tool, or pynetconsole. You can use netconsole and the normal FRC tools to interact with the running robot code.

If you're having problems deploying code to the robot, check out the *troubleshooting section*

Immediate feedback via Netconsole

Note that when you run the deploy command like that, you won't get any feedback from the robot whether your code actually worked or not. If you want to see the feedback from your robot without launching a separate NetConsole

window, a really useful option is `--nc`. This will cause the deploy command to show your program's console output, by launching a netconsole listener.

```
Windows:   py -3 robot.py deploy --nc

Linux/OSX: python3 robot.py deploy --nc
```

Skipping Tests

Now perhaps your tests are failing, but you really need to upload the code, and don't care about the tests. That's OK, you can still upload code to the robot:

```
Windows:   py -3 robot.py deploy --skip-tests

Linux/OSX: python3 robot.py deploy --skip-tests
```

Starting deployed code at boot

If you wish for the deployed code to be started up when the roboRIO boots up, you need to make sure that "Disable RT Startup App" is **not** checked in the roboRIO's web configuration. See the FIRST documentation for more information.

Manually deploying code

Generally, you you just use the steps above. However, if you really want to, then see *How to manually run code*.

Next Steps

Let's talk about *the robot simulator* next.

1.3.4 Robot Simulator

An important (but often neglected) part of developing your robot code is to test it! Because we feel strongly about testing and simulation, the RobotPy project provides tools to make those types of things easier through the pyfrc project.

The pyfrc robot simulator allows very simplistic simulation of your code in real time and displays the results in a (ugly) user interface. To run the simulator, run your robot.py with the following arguments:

```
Windows:   py -3 robot.py sim

Linux/OSX: python3 robot.py sim
```

As there is interest, we will add more features to the simulator. Please feel free to improve it and submit pull requests!

A new feature as of version 2014.7.0 is the addition of showing the robot's simulated motion on a miniature field in the UI. This feature is really useful for early testing of autonomous movements.

Note: For this to work, you must implement a physics module (it's a lot easier than it sounds!). Helper functions are provided to calculate robot position for common drivetrain types (see below for details). There are physics examples provided in the RobotPy Examples Repository for each supported drivetrain type.

Communicating with SmartDashboard

The simulator can be used to communicate with the SmartDashboard or other NetworkTables clients. For this to work, you need to tell SmartDashboard to connect to the IP address that your simulator is listening on (typically this is 127.0.0.1). Using the original SmartDashboard, you need to launch the jar using the following command:

```
$ java -jar SmartDashboard.jar ip 127.0.0.1
```

If you are using the SFX dashboard, there is a configuration option that you can tweak to get it to connect to a different IP. You can also launch it from the command line using the following command:

```
$ java -jar sfx.jar 127.0.0.1
```

Real Joystick support via pygame

If you have pygame installed for Python 3, when you run the simulator any supported joysticks you have plugged in should automatically provide joystick input to the simulator.

Note: Installing pygame requires having a compiler installed, as it has many binary dependencies. It can be a tricky thing to accomplish, please refer to the pygame documentation for the right way to install it for your platform.

New in version 2015.3.6.

Gazebo simulation

This is currently experimental, and hasn't been updated in awhile. If you want to play with it now (and help us fix the bugs!), check out the robotpy-frcsim github repository.

Next Steps

The next section discusses a very important part of writing robot code – *Unit testing robot code*.

1.3.5 Unit testing robot code

pyfrc comes with robot.py extensions that support testing robot code using the py.test python testing tool. To run the unit tests for your robot, just run your robot.py with the following arguments:

```
Windows:   py -3 robot.py test

Linux/OSX: python3 robot.py test
```

Your tests must be in a directory called 'tests' either next to robot.py, or in the directory above where robot.py resides. See 'samples/simple' for an example test program that starts the robot code and runs it through autonomous mode and operator mode.

Builtin unit tests

pyfrc comes with testing functions that can be used to test basic functionality of just about any robot, including running through a simulated practice match. As of pyfrc 2016.1.1, to add these standardized tests to your robot code, you can run the following:

```
Windows:   py -3 robot.py add-tests

Linux/OSX: python3 robot.py add-tests
```

Running this command creates a directory called 'tests' if it doesn't already exist, and then creates a file in your tests directory called pyfrc_test.py, and put the following contents in the file:

```
from pyfrc.tests import *
```

Unlike previous years, all tests work on all types of robots now.

As of pyfrc 2015.0.2, the `--builtin` option allows you to run the builtin tests without needing to create a tests directory.

Writing your own test functions

Often it's useful to create custom tests to test specific things that the generic tests aren't able to test. When running a test, py.test will look for functions in your test modules that start with 'test_'. Each of these functions will be ran, and if any errors occur the tests will fails. A simple test function might look like this:

```
def two_plus(arg):
    return 2 + arg

def test_addition():
    assert two_plus(2) == 4
```

The *assert* keyword can be used to test whether something is True or False, and if the condition is False, the test will fail.

Pytest supports something called a 'fixture', which allows you to add an argument to your test function and it will call the fixture and pass the result to your test function as that argument. pyfrc has a custom pytest plugin that it uses to provide this special functionality to your tests.

For more information:

- RobotPy example code
- py.test documentation

code coverage for tests

pyfrc supports measuring code coverage using the coverage.py module. This feature can be used with any robot.py commands and provide coverage information.

For example, to run the 'test' command to run unit tests:

```
Windows:   py -3 robot.py coverage test

Linux/OSX: python3 robot.py coverage test
```

Or to run coverage over the simulator:

```
Windows:   py -3 robot.py coverage sim

Linux/OSX: python3 robot.py coverage sim
```

Running code coverage while the simulator is running is nice, because you don't have to write unit tests to make sure that you've completely covered your code. Of course, you *should* write unit tests anyways... but this is good for developing code that needs to be run on the robot quickly and you need to make sure that you tested everything first.

When using the code coverage feature, what actually happens is robot.py gets executed *again*, except this time it is executed using the coverage module. This allows coverage.py to completely track code coverage, otherwise any modules that are imported by robot.py (and much of robot.py itself) would not be reported as covered.

Note: There is a py.test module called pytest-cov that is supposed to allow you to run code coverage tests. However, I've found that it doesn't work particularly well for me, and doesn't appear to be maintained anymore.

Note: For some reason, when running the simulation under the code coverage tool, the output is buffered until the process exits. This does not happen under py.test, however. It's not clear why this occurs.

Next Steps

Learn more about some *Best Practices* when creating robot code.

1.3.6 Best Practices

This section has a selection of things that other teams have found to be good things to keep in mind to build robot code that works consistently, and to eliminate possible failures.

If you have things to add to this section, feel free to submit a pull request!

Make sure you're running the latest version of RobotPy!

Seriously. We try to fix bugs as we find them, and if you haven't updated recently, check to see if you're out of date! This is particularly true during build season.

Don't use the print statement/logger excessively

Printing output can easily take up a large proportion of your robot code CPU usage if you do it often enough. Try to limit the amount of things that you print, and your robot will perform better.

Instead, you may want to use this pattern to only print once every half second (or whatever arbitrary period):

```
# Put this in robotInit
self.printTimer = wpilib.Timer()
self.printTimer.start()

..

# Put this where you want to print
if self.printTimer.hasPeriodPassed(0.5):
    self.logger.info("Something happened")
```

Remember, during a competition you can't actually see the output of Netconsole (it gets blocked by the field network), so there's not much point in using these except for diagnostics off the field. In a competition, disable it.

Don't die during the competition!

If you've done any amount of programming in python, you'll notice that it's really easy to crash your robot code – all you need to do is mistype something and BOOM you're done. When python encounters errors (or components such as WPILib or HAL), then what happens is an exception is raised.

Note: If you don't know what exceptions are and how to deal with them, you should read this

---

There's a lot of things that can cause your program to crash, and generally the best way to make sure that it doesn't crash is **test your code**. RobotPy provides some great tools to allow you to simulate your code, and to write unit tests that make sure your code actually works. Whenever you deploy your code using pyfrc, it tries to run your robot code's tests – and this is to try and prevent you from uploading code that will fail on the robot.

However, invariably even with all of the testing you do, something will go wrong during that really critical match, and your code will crash. No fun. Luckily, there's a good technique you can use to help prevent that!

What you need to do is set up a generic exception handler that will catch exceptions, and then if you detect that the FMS is attached (which is only true when you're in an actual match), just continue on instead of crashing the code.

Note: Most of the time when you write code, you never want to create generic exception handlers, but you should try to catch specific exceptions. However, this is a special case and we actually do want to catch all exceptions.

Here's what I mean:

```
try:
    # some code goes here
except:
    if not self.isFmsAttached():
        raise
```

What this does is run some code, and if an exception occurs in that code block, and the FMS is connected, then execution just continues and hopefully everything continues working. However (and this is important), if the FMS is not attached (like in a practice match), then the `raise` keyword tells python to raise the exception anyways, which will most likely crash your robot. But this is good in practice mode – if your driver station is attached, the error and a stack trace should show up in the driver station log, so you can debug the problem.

Now, a naive implementation would just put all of your code inside of a single exception handler – but that's a bad idea. What we're trying to do is make sure that failures in a single part of your robot don't cause the rest of your robot code to not function. What we generally try to do is put each logical piece of code in the main robot loop (teleopPeriodic) in its own exception handler, so that failures are localized to specific subsystems of the robot.

With these thoughts in mind, here's an example of what I mean:

```
def teleopPeriodic(self):

    try:
        if self.joystick.getTrigger():
            self.arm.raise_arm()
    except:
        if not self.isFmsAttached():
            raise
```

```
    try:
        if self.joystick.getRawButton(2):
            self.ball_intake.()
    except:
        if not self.isFmsAttached():
            raise

    # and so on...

    try:
        self.robot_drive.arcadeDrive(self.joystick)
    except:
        if not self.isFmsAttached():
            raise
```

Note: In particular, I always recommend making sure that the call to your robot's drive function is in it's own exception handler, so even if everything else in the robot dies, at least you can still drive around.

Consider using a robot framework

If you're creating anything more than a simple robot, you may find it easier to use a robot framework to help you organize your code and take care of some of the boring details for you. While frameworks sometimes have a learning curve associated with them, once you learn how they work you will find that they can save you a lot of effort and prevent you from making certain kinds of mistakes.

See our documentation on *Robot Code Frameworks*

1.3.7 Using NetworkTables

NetworkTables is a communications protocol used in FIRST Robotics. It provides a simple to use mechanism for communicating information between several computers. There is a single server (typically your robot) and zero or more clients. These clients can be on the driver station, a coprocessor, or anything else on the robot's local control network.

Robot Configuration

Note: These notes apply to all languages that use NetworkTables, not just Python

FIRST introduced the mDNS based addressing for the RoboRIO in 2015, and generally teams that use additional devices have found that while it works at home and sometimes in the pits, it tends to not work correctly on the field at

events. For this reason, if you use pynetworktables on the field, we strongly encourage teams to *ensure every device has a static IP address*.

- Static IPs are `10.XX.XX.2`
- mDNS Hostnames are `roborio-XXXX-frc.local` (don't use these!)

For example, if your team number was 1234, then the static IP to connect to would be `10.12.34.2`.

For information on configuring your RoboRIO and other devices to use static IPs, see the WPILib screensteps documentation.

Server initialization (Robot)

WPILib automatically starts NetworkTables for you, and no additional configuration should need to be done.

Client initialization (Driver Station/Coprocessor)

Note: For install instructions, see *pynetworktables installation instructions*

As of 2017, this is very easy to do. Here's the code:

```
from networktables import NetworkTables

NetworkTables.initialize(server='10.xx.xx.2')
```

The key is specifying the correct server hostname. See the above section on robot configuration for this.

Theory of operation

In its most abstract form, NetworkTables can be thought of as a dictionary that is shared across multiple computers across the network. As you change the value of a table on one computer, the value is transmitted to the other side and can be retrieved there.

The keys of this dictionary **must** be strings, but the values can be numbers, strings, booleans, various array types, or raw binary. Strictly speaking, the keys can be any string value, but they are typically path-like values such as `/SmartDashboard/foo`.

When you call `NetworkTables.getTable`, this retrieves a `NetworkTable` instance that allows you to perform operations on a specified path:

```
table = NetworkTables.getTable('SmartDashboard')

# This retrieves a boolean at /SmartDashboard/foo
foo = table.getBoolean('foo', True)
```

There is also an concept of subtables:

```
subtable = table.getSubTable('bar')

# This retrieves /SmartDashboard/bar/baz
baz = table.getNumber('baz', 1)
```

As you may have guessed from the above example, once you obtain a NetworkTable instance, there are very simple functions you can call to send and receive NetworkTables data.

- To retrieve values, call `table.getXXX(name, default)`
- To send values, call `table.putXXX(name, value)`

NetworkTables can also be told to call a function when a particular key in the table is updated.

Code Samples

There are many code samples showing various aspects of NetworkTables operation. See the pynetworktables examples page.

See also:

NetworkTables API Reference

Troubleshooting

See also:

pynetworktables troubleshooting

External tools

WPILib's OutlineViewer (requires Java) is a great tool for connecting to networktables and seeing what's being transmitted.

1.3.8 Image Processing

This is a more advanced topic, and nobody has taken the time to write this section yet. Will you be the one to do so?

1.3.9 Example Code

Sometimes the documentation just isn't enough. To help you get started, the RobotPy project provides many example programs that can be a good starting point.

- Robot Code examples
- NetworkTables samples

1.4 Robot Code Frameworks

After creating code for a few robots, you'll notice that there are a lot of similarities between the code. Robot code frameworks are a collection of patterns and ideas that are generally useful for creating robot code.

While frameworks sometimes have a learning curve associated with them, once you learn how they work you will find that they can save you a lot of effort and prevent you from making certain kinds of mistakes.

- *Command Framework*: this framework comes with WPILib
- *MagicBot Framework*: Created as a pythonic alternative to the Command framework

1.4.1 Command Framework

If you're coming from C++ or Java, you are probably familiar with the Command based robot paradigm. All of the pieces you're used to are still here, but this guide might help save you a bit of time as you make the transition.

If you're starting Command based programming in Python and have no experience with it in other languages, make sure you familiarize yourself with it before proceeding. This guide only covers differences between Python and the other languages's versions of that paradigm.

For the impatient, a fully-working example program is available. You can start with that and modify it to meet your needs.

The Basics

The structure of a Command based program is simple and predictable. You inherit from the `IterativeRobot` class, configure the robot in `robotInit()`, and then run the `Scheduler` inside the `*Periodic()` methods.

Writing it can be done rather quickly, but the robotpy-wpilib-utilities package contains a pre-built skeleton class you can inherit, meaning that your program only needs to implement functions unique to your robot. Here is an example:

```
import wpilib
from commandbased import CommandBasedRobot

from commands import AutonomousCommandGroup

class MyRobot(CommandBasedRobot):

    def robotInit(self):
        '''Initialize things like subsystems'''

        self.autonomous = AutonomousCommandGroup()


    def autonomousInit(self):
        self.autonomous.start()
```

Setting up the scheduler and running it are handled by the CommandBasedRobot class. You only need to write the code for `*Init()` methods you want to use. Since you are overriding that class, you can easily change any functionality that doesn't work for your robot, though the default implementation should work for most cases.

Error Handling

Crashes happen. Even the most careful programmer can write a command that breaks under unexpected conditions. Normally this will cause your program to reboot, costing precious seconds during a competition. With this in mind, the `Scheduler` is run inside an exception handler. If a crash happens inside a Command while your robot is connection to the Field Management System (i.e. - during a competition) the exception will be caught, running commands will be canceled, the error will be printed on the driver's station, and the Scheduler loop will continue running normally.

If you need more advanced error handling functionality, you can override the `'handleCrash()'` method in your robot.py.

Pythonic Command Based Programming

All the classes you know from C++ and Java still exists in Python, allowing you to directly port your code with minimal changes. However, you can use some of the advantages of Python to make a few things a bit easier.

Subsystems

In C++ and Java, the recommended way of making subsystems available to Commands is to instantiate them in the `init()` method of a class that subclasses `Command`, and then use that as the base class for all of your classes. Even in those languages that can be a bit unwieldy, especially if you want your commands to inherit from multiple base classes.

A more appropriate method in Python is to instantiate your subsystems inside a module, then import that module anywhere you need subsystems. Here is a simple example of that module, which we will call `subsystems`:

```
from subsystemtype import SubsystemType

subsystem1 = None

def init():
    global subsystem1

    subsystem1 = SubsystemType()
```

You can import this module in your `robot.py` and then call `subsystems.init()` inside `robotInit()` before any commands are instantiated. Then you can access your subsystem from any Command like this:

```
import subsystems
from wpilib.command import InstantCommand

class ExampleCommand(InstantCommand):

    def __init__(self):
        self.requires(subsystems.subsystem1)

    def initialize(self):
        subsystems.subsystem1.do_something()
```

By using this method you can override any Command provided by WPILib or robotpy-wpilib-utilities, with pythonic namespacing. For even better structure, make `subsystems` a package that holds the code for all of your subsystems, as demonstrated in the example program.

RobotMap

Having a single place to store your robot's configuration can be very helpful, and this is why most Command based robots integrate a `RobotMap.*` file to store port numbers. In Python you can create a `robotmap` module that will act similarly. There are many different possible ways to manage your ports:

1.) Raw variables:

```
drive_front_left = 1
drive_front_right = 2
drive_rear_left = 3
drive_rear_right = 4
```

2.) Dictionary:

```
drive = {
    'front_left': 1,
    'front_right': 2,
    'rear_left': 3,
    'rear_right': 4
}
```

3.) Object Properties:

```
class PortList():
    pass

drive = PortList()

drive.front_left = 1
drive.front_right = 2
drive.rear_left = 3
drive.rear_right = 4
```

Whichever method you choose, you can utilize it simply by importing:

```
import robotmap
from wpilib.command import Subsystem

class DriveSubsystem(Subsystem):
    def __init__():
        front_left_motor = robotmap.drive_front_left
```

Final Thoughts

Welcome to FRC programming with Python. This documentation is still developing, so if you find a great trick to make programming your robot in the Command based paradigm more "pythonic", please update it with your ideas.

See also:

MagicBot Framework

1.4.2 MagicBot Framework

MagicBot is an opinionated framework for creating Python robot programs for the FIRST Robotics Competition. It is envisioned to be an easier to use pythonic alternative to the Command framework – but it's not quite there yet.

While MagicBot will tend to be more useful for complex multi-module programs, it does remove some of the boiler-plate associated with simple programs as well.

Philosophy

You should use the `MagicRobot` class as your base robot class. You'll note that it's similar to `IterativeRobot`:

```
import magicbot
import wpilib

class MyRobot(magicbot.MagicRobot):

    def createObjects(self):
        '''Create motors and stuff here'''
        pass

    def teleopInit(self):
        '''Called when teleop starts; optional'''

    def teleopPeriodic(self):
        '''Called on each iteration of the control loop'''
```

```
if __name__ == '__main__':
    wpilib.run(MyRobot)
```

A robot control program can be divided into several logical parts (think drivetrain, forklift, elevator, etc). We refer to these parts as "Components".

Components

When you design your robot code, you should define each of the components of your robot and order them in a hierarchy, with "low level" components at the bottom and "high level" components at the top.

- "Low level" components are those that directly interact with physical hardware: drivetrain, elevator, grabber
- "High level" components are those that only interact with other components: these are generally automatic behaviors or encapsulation of multiple low level components into an easier to use component

Generally speaking, components should never interact with operator controls such as joysticks. This allows the components to be used in autonomous mode and in teleoperated mode.

Components should have three types of methods (excluding internal methods):

- Control methods
- Informational methods
- An `execute` method

Control methods

Think of these as 'verb' functions. In other words, calling one of these means that you want that particular thing to happen.

Control methods store information necessary to perform a desired action, but **do not actually execute the action**. They are generally called either from `teleopPeriodic`, another component's control method, or from an autonomous mode.

Example method names: raise_arm, lower_arm, shoot

Informational methods

These are basic methods that tell something about a component. They are typically called from control methods, but may be called from execute as well.

Example method names: is_arm_lowered, ready_to_shoot

execute method

The `execute` method reads the data stored by the control methods, and then sends data to output devices such as motors to execute the action. You should not call the execute function as `execute` is automatically called by MagicRobot if you define it as a magic component.

Component creation

Components are instantiated by the MagicRobot class. You can tell the `MagicRobot` class to create magic components by defining the variable names and types in your MyRobot object.

```
from components import Elevator, Forklift

class MyRobot(MagicRobot):
```

```
    elevator = Elevator
    forklift = Forklift

    def teleopPeriodic(self):

        # self.elevator is now an instance of Elevator
```

Variable injection

To reduce boilerplate associated with passing components around, and to enhance autocomplete for PyDev, MagicRobot can inject variables defined in your robot class into other components, and autonomous modes. Check out this example:

```
class MyRobot(MagicRobot):

    def createObjects(self):
        self.elevator_motor = wpilib.Talon(2)


class Elevator:

    elevator_motor = wpilib.Talon

    def execute(self):
        # self.elevator_motor is a reference to the Talon instance
        # created in MyRobot.createObjects
```

Operator Control code

Code that controls components should go in the `teleopPeriodic` method. This is really the only place that you should generally interact with a Joystick or NetworkTables variable that directly triggers an action to happen.

To ensure that a single portion of robot code cannot bring down your entire robot program during a competition, MagicRobot provides an `onException` method that will either swallow the exception and report it to the Driver Station, or if not connected to the FMS will crash the robot so that you can inspect the error:

```
try:
    if self.joystick.getTrigger():
        self.component.doSomething()
except:
    self.onException()
```

MagicRobot also provides a `consumeExceptions` method that you can wrap your code with using a `with` statement instead:

```
with self.consumeExceptions():
    if self.joystick.getTrigger():
        self.component.doSomething()
```

Note: Most of the time when you write code, you never want to create generic exception handlers, but you should try to catch specific exceptions. However, this is a special case and we actually do want to catch all exceptions.

See also:

RobotPy Guidelines

Autonomous mode

MagicBot supports loading multiple autonomous modes from a python package called 'autonomous'. To create this package, you must:

- Create a folder called 'autonomous' in the same directory as robot.py
- Add an empty file called '__init__.py' to that folder

Any `.py` files that you add to the autonomous package will automatically be loaded at robot startup.

See also:

`AutonomousModeSelector` on how to define an autonomous mode.

Dashboard & coprocessor communications

The simplest method to communicate with other programs external to your robot code (examples include dashboards and image processing code) is using NetworkTables. NetworkTables is a distributed keystore, or put more simply, it is similar to a python dictionary that is shared across multiple processes.

Note: For more information about NetworkTables, see *Using NetworkTables*

Magicbot provides a simple way to interact with NetworkTables, using the `tunable` property. It provides a python property that has get/set functions that read and write from NetworkTables. The NetworkTables key is automatically determined by the name of your object instance and the name of the attribute that the tunable is assigned to.

In the following example, this would create a NetworkTables variable called */components/mine/foo*, and assign it a default value of 1.0:

```
class MyComponent:

    foo = tunable(default=1.0)

...

class MyRobot:
    mine = MyComponent
```

To access the variable, in `MyComponent` you can read or write `self.foo` and it will read/write to NetworkTables.

For more information about creating custom dashboards, see the following:

- pynetworktables2js docs
- Smartdashboard docs

Example Components

Low level components

Low level components are those that directly interact with hardware. Generally, these should not be stateful but should express simple actions that cause the component to do whatever it is in a simple way, so when it doesn't work you can bypass any automation and more easily test the component.

Here's an example single-wheel shooter component:

```
class Shooter:

    shooter_motor = wpilib.Talon

    # speed is tunable via NetworkTables
    shoot_speed = tunable(1.0)

    def __init__(self):
        self.enabled = False

    def enable(self):
        '''Causes the shooter motor to spin'''
        self.enabled = True

    def is_ready(self):
        # in a real robot, you'd be using an encoder to determine if the
        # shooter were at the right speed..
        return True

    def execute(self):
        '''This gets called at the end of the control loop'''
        if self.enabled:
            self.shooter_motor.set(self.shoot_speed)
        else:
            self.shooter_motor.set(0)

        self.enabled = False
```

Now, this is useful, but you'll note that it's not particularly smart. It just makes the component work. Which is great – very easy to debug. Let's automate some stuff now.

High level components

High level components are those that control other components to automate one or more of them for automated behaviors. Consider the example of the Shooter component above – let's say that you have some intake component that needs to feed a ball into the shooter when the shooter is ready. At that point, you're ready for high level components! First, let's just define what the low-level intake interface is:

- Has a function 'feed_shooter' which will send the ball to the shooter

Let's automate these two using a state machine helper:

```
from magicbot import StateMachine, state, timed_state

class ShooterControl(StateMachine):
    shooter = Shooter
    intake = Intake

    def fire(self):
        '''This function is called from teleop or autonomous to cause the
           shooter to fire'''
        self.engage()

    @state(first=True)
    def prepare_to_fire(self):
        '''First state -- waits until shooter is ready before going to the
           next action in the sequence'''
        self.shooter.enable()
```

```
        if self.shooter.is_ready():
            self.next_state_now('firing')

    @timed_state(duration=1, must_finish=True)
    def firing(self):
        '''Fires the ball'''
        self.shooter.enable()
        self.intake.feed_shooter()
```

There's a few special things to point out here:

- There are two steps in this state machine: 'prepare_to_fire' and 'firing'. The first step is 'prepare_to_fire', and it only transitions into 'firing' if the shooter is ready.
- When you want the state machine to start executing, you call the 'engage' method. Of course, it's nice to have a semantically useful name, so we defined a function called 'fire' which just calls the 'engage' function for us.
- True to magicbot philosophy, the state machine will only execute if the 'engage' function is continuously called. So if you call engage, then prepare_to_fire will execute. But if you neglect to call engage again, then no states will execute.

Note: There is an exception to this rule! Once you start firing, if the intake stops then the ball will get stuck, so we *must* continue even if engage doesn't occur. To tell the state machine about this, we pass the `must_finish` argument to @timed_state which will continue executing the state machine step until the duration has expired.

Now obviously this is a very simple example, but you can extend the sequence of events that happens as much as you want. It allows you to specify arbitrarily complex sets of steps to happen, and the resulting code is really easy to understand.

Using these components

Here's one way that you might put them together in your robot.py file:

```
class MyRobot(magicbot.MagicRobot):

    # High level components go first
    shooter_control = ShooterControl

    # Low level components come last
    intake = Intake
    shooter = Shooter

    ...

    def teleopPeriodic(self):

        if self.joystick.getTrigger():
            self.shooter_control.fire()
```

API Reference

See also:

Magicbot API Reference

1.5 Hardware & Sensors

FIRST has put together a lot of great documentation that can tell you how to connect hardware devices and interact with it from robot code.

- Using actuators (motors, servos, and relays)
- Using CAN Devices
- WPILib Sensors
- Driver Station Inputs and Feedback

While their documentation code samples are in C++ and Java, it's fairly straightforward to translate them to python – RobotPy includes support for all components that are supported by WPILib's Java implementation, and generally the objects have the same name and method names.

If you have problems translating their code samples into Python, you can use our support resources to get help (see *Support*).

1.6 Troubleshooting

1.6.1 Robot Code

Problem: I can't run code on the robot!

There are lots of things that can go wrong here. It is very important to have the latest versions of the FIRST robot software installed:

- Robot Image
- Driver Station + Tools

The FIRST screensteps documentation contains information on what the current versions are, and how to go about updating the software.

You should also have the latest version of the RobotPy software packages:

- Do you have the latest version of pyfrc?

Warning: Make sure that the version of WPILib on your computer matches the version installed on the robot! You can check what version you have locally by running:

```
Windows: py -3 -m pip list

Linux/OSX: pip3 list
```

1. Did you run the deploy command to put the code on the robot?
2. Make sure you have the latest version of pyfrc! Older versions **won't** work.
3. Read any error messages that pyfrc might give you. They might be useful. :)

Problem: no module named 'wpilib'

If you're on your local computer, did you *install pyfrc via pip*?

If you're on the roboRIO, did you *install RobotPy*?

Problem: pyfrc cannot connect to the robot, or appears to hang

1. Can you ping your robot from the machine that you're deploying code from? If not, pyfrc isn't going to be able to connect to the robot either.
2. Try to ssh into your robot, using PuTTY or the `ssh` command on Linux/OSX. The username to use is `lvuser`, and the password is an empty string. If this doesn't work, pyfrc won't be able to copy files to your robot
3. If all of that works, it might just be that you typed the wrong hostname to connect to. There's a file called `.deploy_cfg` next to your `robot.py` that pyfrc created. Delete it, and try again.

Problem: I deploy successfully, but the driver station still shows 'No Robot Code'

1. Did you use the `--nc` option to the deploy command? Your code may have crashed, and the output should be visible on netconsole.
2. If you can't see any useful output there, then ssh into the robot and run `ps -Af | grep python3`. If nothing shows up, it means your python code crashed and you'll need to debug it. Try running it manually on the robot using this command:

```
python3 /home/lvuser/py/robot.py run
```

Problem: When I run deploy, it complains that the WPILib versions don't match

Not surprisingly, the error message is correct.

During deployment, pyfrc does a number of checks to ensure that your robot is setup properly for running python robot code. One of these checks is testing the WPILib version number against the version installed on your computer (it's installed when you install pyfrc).

You should either:

- Upgrade the RobotPy installation on the robot to match the newer version on your computer. See the RobotPy install guide for more info.
- Upgrade the pyfrc installation on your computer to match the version on the robot. Just run:

```
    pip3 install pyfrc --upgrade
```

If you *really* don't want pyfrc to do the version check and need to deploy the code *now*, you can specify the `--no-version-check` option. However, this isn't recommended.

1.6.2 pynetworktables

Ensure you're using the correct mode

If you're running pynetworktables as part of a RobotPy robot – relax, pynetworktables is setup as a server automatically for you, just like in WPILib!

If you're trying to connect to the robot from a coprocessor (such as a Raspberry Pi) or from the driver station, then you will need to ensure that you initialize pynetworktables correctly.

Thankfully, this is super easy as of 2017. Here's the code:

```
from networktables import NetworkTables

NetworkTables.initialize(server='10.xx.xx.2')
```

Don't know what the right hostname is? That's what the next section is for...

Use static IPs when using pynetworktables

See also:

Using NetworkTables

Problem: I can't determine if networktables has connected

Make sure that you have enabled python logging (it's not enabled by default):

```
# To see messages from networktables, you must setup logging
import logging
logging.basicConfig(level=logging.DEBUG)
```

Once you've enabled logging, look for messages that look like this:

```
INFO:nt:CONNECTED 10.14.18.2 port 40162 (...)
```

If you see a message like this, it means that your client has connected to the robot successfully. If you don't see it, that means there's still a problem. Usually the problem is that you set the hostname incorrectly in your call to `NetworkTables.initialize`.

1.7 Support

The RobotPy project was started in 2010, and since then the community surrounding RobotPy has continued to grow! If you have questions about how to do something with RobotPy, you can ask questions in the following locations:

- RobotPy mailing list
- ChiefDelphi Python Forums

We have found that most problems users have are actually questions generic to WPILib-based languages like C++/Java, so searching around the ChiefDelphi forums could be useful if you don't have a python-specific question.

During the FRC build season, you can probably expect answers to your questions within a day or two if you send messages to the mailing list. As community members are also members of FRC teams, you can expect that the closer we get to the end of the build season, the harder it will be for community members to respond to your questions!

1.7.1 Reporting Bugs

If you run into a problem with RobotPy that you think is a bug, or perhaps there is something wrong with the documentation or just too difficult to do, please feel free to file bug reports on the github issue tracker. Someone should respond within a day or two, especially during the FIRST build season.

1.7.2 Contributing new fixes or features

RobotPy is intended to be a project that all members of the FIRST community can **quickly** and **easily** contribute to. If you find a bug, or have an idea that you think others can use:

1. Fork the appropriate git repository to your github account
2. Create your feature branch (*git checkout -b my-new-feature*)
3. Commit your changes (*git commit -am 'Add some feature'*)
4. Push to the branch (*git push -u origin my-new-feature*)
5. Create new Pull Request on github

Github has a lot of documentation about forking repositories and pull requests, so be sure to check out those resources.

1.7.3 RobotPy Chat

During the FRC Build Season, some RobotPy developers may be able to be reached on the RobotPy Gitter Channel.

Note: the channel is not very active, but if you stick around for a day or two someone will probably answer your question – think in terms of email response time

The channel tends to be most active between 11pm and 1am EST.

1.8 Developer Documentation

These pages contain information about various internal details of RobotPy which are useful for advanced users and those interested in developing RobotPy itself. We will endeavor to keep these pages up to date. :)

1.8.1 Design

Goals

The python implementation of WPILib/HAL is derived from the Java implementation of WPILib. In particular, we strive to keep the python implementation of WPILib as close to the spirit of the original WPILib java libraries as we can, only adding language-specific features where it makes sense.

Things that you won't find in the original WPILib can be found in the _impl package.

If you have a suggestion for things to add to WPILib, we suggest making a request to the WPILib team directly, or if appropriate you can add it to the robotpy_ext package, which is a separate package for "high quality code of things that should be in WPILib, but aren't". This package is installed by the RobotPy installer by default.

HAL Loading

Currently, the HAL is split into two python packages:

- hal - Provided by the robotpy-hal-base package
- hal_impl - Provided by either robotpy-hal-roborio or robotpy-hal-sim

You can only have a single hal_impl package installed in a particular python installation.

The `hal` package provides the definition of the functions and various types & required constants.

The `hal_impl` package provides the actual implementation of the HAL functions, or links them to a shared DLL via ctypes.

Adding options to robot.py

When `run()` is called, that function determines available commands that can be run, and parses command line arguments to pass to the commands. Examples of commands include:

- Running the robot code
- Running the robot code, connected to a simulator
- Running unit tests on the robot code
- And lots more!

python setuptools has a feature that allows you to extend the commands available to robot.py without needing to modify WPILib's code. To add your own command, do the following:

- Define a setuptools entrypoint in your package's setup.py (see below)
- The entrypoint name is the command to add
- **The entrypoint must point at an object that has the following properties:**
  - Must have a docstring (shown when `--help` is given)
  - Constructor must take a single argument (it is an argparse parser which options can be added to)
  - Must have a 'run' function which takes two arguments: options, and robot_class. It must also take arbitrary keyword arguments via the `**kwargs` mechanism. If it receives arguments that it does not recognize, the entry point must ignore any such options.

If your command's run function is called, it is your command's responsibility to execute the robot code (if that is desired). This sample command demonstrates how to do this:

```
class SampleCommand:
    '''Help text shown to user'''

    def __init__(self, parser):
        pass

    def run(self, options, robot_class, **static_options):
```

```
        # runs the robot code main loop
        robot_class.main(robot_class)
```

To register your command as a robotpy extension, you must add the following to your setup.py setup() invocation:

```
from setuptools import setup

setup(
      ...
      entry_points={'robotpy': ['name_of_command = package.module:CommandClassName']},
      ...
      )
```

1.8.2 Developer Installation

Installing WPILib from git

Warning: These instructions are only intended for those users wanting to deploy a custom modified version of the RobotPy WPILib source code

1. Install python on the roboRIO using one of the methods in the *installation guide*.
2. Checkout the robotpy-wpilib git repository, and make your changes there
3. To deploy your changes, you can run `devtools/build_and_deploy.sh` from the root of the git repository.

1.8.3 Deploy process details

When the code is uploaded to the robot, the following steps occur:

- SSH/sftp operations are performed as the `lvuser` user (this is REALLY important, don't use the `admin` user!)
- pyfrc does some checks to make sure the environment is setup properly
- The directory containing `robot.py` is recursively copied to the the directory `/home/lvuser/py`
- The files `robotCommand` and `robotDebugCommand` are created
- `/usr/local/frc/bin/frcKillRobot.sh -t -r` is called, which causes any existing robot code to be killed, and the new code is launched

If you wish for the code to be started up when the roboRIO boots up, you need to make sure that "Disable RT Startup App" is **not** checked in the roboRIO's web configuration.

These steps are compatible with what C++/Java does when deployed by eclipse, so you should be able to seamlessly switch between python and other FRC languages!

How to manually run code

Note: Generally, you shouldn't need to use this process.

If you don't have (or don't want) to install pyfrc, running code manually is pretty simple too.

1. Make sure you have RobotPy installed on the robot

2. Use scp or sftp (Filezilla is a great GUI product to use for this) to copy your robot code to the roboRIO
3. ssh into the roboRIO, and run your robot code manually

```
python3 robot.py run
```

Your driver station should be able to connect to your code, and it will be able to operate your robot!

Note: This is good for running experimental code, but it won't start the code when the robot starts up. Use pyfrc to do that.

1.8.4 Updating RobotPy source code to match WPILib

Every year, the WPILib team makes improvements to WPILib, so RobotPy needs to be updated to maintain compatibility. While this is largely a manual process, we now use a tool called git-source-track to assist with this process.

Note: git-source-track only works on Linux/OSX at this time. If you're interested in helping with the porting process and you use Windows, file a github issue and we'll try to help you out.

Using git-source-track

First, you need to checkout the git repo for allwpilib and the RobotPy WPILib next to each other in the same directory like so:

```
allwpilib/
robotpy-wpilib/
```

The way git-source-track works is it looks for a comment in the header of each tracked file that looks like this:

```
# validated: 2015-12-24 DS 6d854af athena/java/edu/wpi/first/wpilibj/Compressor.java
```

This stores when the file was validated to match the original source, initials of the person that did the validation, what commit it was validated against, and the path to the original source file.

Finding differences

From the *robotpy-wpilib* directory, you can run `git source-track` and it will output all of the configured files and their status. The status codes include:

- `OK`: File is up to date, no changes required
- `OLD`: The tracked file has been updated, `git source-track diff FILENAME` can be used to show all of the git log messages and associated diffs.
- `ERR`: The tracked file has moved or has been deleted
- `--`: The file is not currently being tracked

Sometimes, commits are added to WPILib which only change comments, formatting, or mass file renames – these don't change the semantic content of the file, so we can ignore those commits. When identified, those commits should be added to `devtools/exclude_commits`.

Looking at differences

Once you've identified a file that needs to be updated, then you can run:

```
git source-track diff FILENAME
```

This will output a verbose git log command that will show associated commit messages and the diff output associated with that commit for that specific file. Note that it will only show the change for that specific file, it will not show changes for other files (use `git log -p COMMITHASH` in the original source directory if you want to see other changes).

After running `git source-track diff` it will ask you if you want to validate the file. If no python-significant changes have been made, then you can answer 'y' and the validation header will be updated.

Adding new files

Unfortunately, git-source-track doesn't currently have a mechanism that allows it to identify new files that need to be ported. We need to do that manually.

Dealing with RobotPy-specific files

We don't need to track those files; `git source-track set-notrack FILENAME` takes care of it.

After you finish porting the changes

Once you've finished making the appropriate changes to the python code, then you should update the validation header in the source file. Thankfully, there's a command to do this:

```
git source-track set-valid FILENAME
```

It will store the current date and the tracked git commit.

Additionally, if you answer 'y' after running `git source-track diff FILENAME`, then it will update the validation header in the file.

HAL Changes

If there are changes to the HAL, we have some scripts that should be able to help out here.

- `devtools/hal_fix.sh`: This detects errors in the HAL, and if you pass it the `--stubs` argument it can print out the correct HAL definitions or a python stub. Use `--help` for more information.

Syntax/Style Guide

Except where it makes sense, developers should try to retain the structure and naming conventions that the Java implementation of WPILib follows. There are a few guidelines that can be helpful when translating Java to Python:

- Member variables such as `m_foo` should be converted to `self.foo`
- Private/protected functions (but NOT variables) should start with an underscore
- Always retain original javadoc documentation, and convert it to the appropriate standard python docstring (see below)

Converting javadocs to docstrings

There is an HTML page in devtools called `convert_javadoc.html` that you can use. The way it works is you copy a Java docstring in the top box (you can also paste in a function definition too) and it will output a python docstring in the bottom box. When adding new APIs that have documentation, this tool is invaluable and will save you a ton of time – but feel free to improve it!

Enums

Though Python 3 does have an enum module, we currently are not using it to implement the enums found in the Java WPILib (this may change in the future). Instead, we define a class that has static variables with appropriate names and assign integers appropriately:

```
class SomeObject:

    class MyEnum:
        VALUE1 = 1
        VALUE2 = 2
```

Many WPILib classes define various enums, see existing code for example translations.

Synchronized

The python language has no equivalent to the Java `synchronized` keyword. Instead, create a `threading.RLock` instance object called `self.lock`, and surround the internal function body with a `with self.lock:` block:

```
def someSynchronizedFunction(self):
    with self.lock:
        # do something here...
```

Interfaces

While we define the various interfaces for documentation's sake, the Python WPILib does not actually utilize most of the interfaces.

Final thoughts

Before translating WPILib Java code to RobotPy's WPILib, first take some time and read through the existing RobotPy code to get a feel for the style of the code. Try to keep it pythonic and yet true to the original spirit of the code. Style *does* matter, as students will be reading through this code and it will potentially influence their decisions in the future.

Remember, all contributions are welcome, no matter how big or small!

CHAPTER 2

Indices and tables

- genindex
- modindex
- search