RobotPy Documentation

Release 2023

RobotPy development team

Jan 07, 2023

ROBOT PROGRAMMING

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.

Please read our *Upgrade Notes* page for things that have changed this season that you should be aware of.

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

We are working with the WPILib team to make Python an officially supported language in 2024. See https://wpilib.org/blog/bringing-python-to-frc for the announcement.

CHAPTER

ONE

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:

- pyntcore: python bindings for NetworkTables that you can use to communcate with a dashboard and/or your robot.
- pynetworktables: legacy NetworkTables implementation that you can use to communcate with SmartDashboard and/or your robot.
- pynetconsole: A simple netconsole implementation in python
- robotpy-cscore: Python bindings for cscore, a powerful camera/streaming library
- robotpy-apriltag: Python bindings for the WPILib apriltag library
- 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 C++ 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 RobotPy on your computer*
- 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 Upgrade Notes

Here are major things that have changed for the 2023 season:

- See below for NT4 related changes
- cscore is much easier to build now, and we distribute wheels for Windows/Linux/macOS
- robotpy-commands is no longer supported, only commands v2
- There are two new packages: robotpy-apriltag and robotpy-wpinet

1.2.1 NetworkTables 4 (NT4)

pynetworktables will not be upgraded to support NT4. We will still fix bugs in its NT3 support, but we recommend all users to switch to pyntcore.

pyntcore now provides a similar API to networktables, but it lives in the `ntcore` package now.

1.2.2 Linux specific notes

Linux requires Ubuntu 22.04 or a distribution with an equivalent (or newer) glibc installation. See *linux installation page* for more information.

1.3 Installation

Writing Robot code in Python:

- If you wish to deploy code or use the Robot Simulator, see the *computer Installation documentation*.
- To install RobotPy on your robot, see the *RobotPy installation documentation*.

Using NetworkTables from Python:

- To install pynetworktables on a system that does not have RobotPy or pyfrc installed on it (such as a coprocessor like the Raspberry Pi), see the *pynetworktables installation documentation*.

Using cscore from Python:

- See the *robotpy-cscore installation documentation*.

Installing extra software packages on the RoboRIO:

- You can install third party packages such as OpenCV, NumPy, robotpy-cscore, robotpy-ctre, and other python packages on your RoboRIO using the *RobotPy Installer*.

1.3.1 RobotPy Components

RobotPy provides a meta installation package that makes it simpler to install and upgrade RobotPy. The meta package allows you to run `pip install robotpy` and this install all of the core RobotPy packages. This meta package is used both for installation on your computer and on your robot.

Optional/Vendor Components

If you install just the `robotpy` package via pip, then all of the core RobotPy wrappers around WPILib will be installed. However there are several groups of optional components that you can install. Vendor categories:

- `ctre` - Cross The Road Electronics motor controllers
- `navx` - Kauai Labs NavX MXP Robotics Navigation
- `photonvision` - PhotonVision computer vision vendor library
- `pathplannerlib` - PathPlannerLib path planning vendor library
- `photonvision` - PhotonVision computer vision vendor library
- `rev` - REV Robotics motor controllers and color sensors

Optional WPILib component categories:

- `apriltag` - WPILib apriltag library
- `commands2` - WPILib Commands framework (2020+)
- `cscore` - WPILib cscore library
- `romi` - Romi robot specific components
- `sim` - WPILib extra simulation support

Install all vendors and WPILib optional components:

- `all`

Using components

Component categories are represented as 'extra requirements' for the RobotPy package. Pip allows you to install extra requirements by putting the names of the categories in brackets.

Let's say that you wanted to install the latest version of the NavX software along with command based programming. You would do this

Windows

```
py -3 -m pip install -U robotpy[navx,commands]
```

Linux/macOS

```
pip3 install -U robotpy[navx,commands]
```

Or if you wanted to install everything:

Windows

```
py -3 -m pip install -U robotpy[all]
```

Linux/macOS

```
pip3 install -U robotpy[all]
```

RoboRIO vs Computer

The RobotPy meta package is used for installation on both the RoboRIO and on your computer.

1.3.2 Computer Installation

Note: installation via pip typically requires internet access

RobotPy requires Python 3.7/3.8/3.9/3.10/3.11 to be installed on your computer. We no longer support 32-bit installations of Python, you must have a 64-bit python installed.

- Python for Windows
- Python for macOS

Once you have installed Python, you can use pip to install RobotPy. While it is possible to install without pip, due to the large number of dependencies this is not recommended nor is it supported.

Windows

Warning: On Windows, the Visual Studio 2019 redistributable package is required to be installed.

Run the following command from cmd or Powershell to install the core RobotPy packages:

```
py -3 -m pip install robotpy
```

See also:

This command only installs the core RobotPy packages. See additional details for installing *optional/vendor components*

To upgrade, you can run this:

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

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 robotpy
```

macOS

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

```
pip3 install robotpy
```

See also:

This command only installs the core RobotPy packages. See additional details for installing *optional/vendor components*

To upgrade, you can run this:

```
pip3 install --upgrade robotpy
```

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 robotpy
```

Linux

Since 2021, RobotPy distributes manylinux binary wheels on PyPI. However, installing these requires a distro that has glibc 2.35 or newer, and an installer that implements **PEP 600**, such as pip 20.3 or newer. You can check your version of pip with the following command:

```
pip3 --version
```

If you need to upgrade your version of pip, it is highly recommended to use a virtual environment.

If you have a compatible version of pip, you can simply run:

```
pip3 install robotpy
```

See also:

This command only installs the core RobotPy packages. See additional details for installing *optional/vendor components*

To upgrade, you can run this:

```
pip3 install --upgrade robotpy
```

The following Linux distributions are known to work, but this list is not necessarily comprehensive:

- Ubuntu 22.04+
- Fedora 36+

- Arch Linux

If you manage to install the packages and get the following error or something similar, your system is most likely not compatible with RobotPy:

```
OSError: /usr/lib/x86_64-linux-gnu/libstdc++.so.6: version `GLIBCXX_3.4.22' not found␣
↪(required by /usr/local/lib/python3.7/dist-packages/wpiutil/lib/libwpiutil.so)
```

source install

Alternatively, if you have a C++17 compiler installed, you may be able to use pip to install RobotPy from source.

> **Warning:** It may take a very long time to install!

> **Warning:** Mixing our pre-built wheels with source installs may cause runtime errors. This is due to internal ABI incompatibility between compiler versions.
>
> Our wheels are built on Ubuntu 22.04 with GCC 11.

If you need to build with a specific compiler version, you can specify them using the `CC` and `CXX` environment variables:

```
export CC=gcc-12 CXX=g++-12
```

1.3.3 Robot Installation

These instructions will help you get RobotPy installed on your RoboRIO, which will allow you to write robot code using Python. If you install RobotPy on your RoboRIO, you are still able to deploy C++ and Java programs without any conflicts.

Note: If you're looking for instructions to use NetworkTables from Python, you probably want the *pynetworktables installation documentation*.

Install requirements

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

RobotPy is truly cross platform, and can be installed from Windows, most Linux distributions, and from Mac macOS also. To install/use the installer, you must have Python 3.7+ installed. You should install the installer via pip (requires internet access) by installing the core RobotPy components (see the *computer installation* section for more details).

Windows

```
py -3 -m pip install robotpy
```

Linux/macOS

```
pip3 install robotpy
```

Install process

The RoboRIO robot controller is typically not connected to a network that has internet access, so there are two stages to installing RobotPy.

- First, you need to connect your computer to the internet and use the installer to download the packages to your computer.
- Second, disconnect from the internet and connect to the network that the RoboRIO is on

The details for each stage will be discussed below. You can run the installer via python. This is slightly different on Windows/macOS/Linux.

Install Python on a roboRIO

Note: This step only needs to be done once.

Installing Python and the RobotPy packages are separated into two different steps. Once you are connected to the internet, you can run this to download Python for roboRIO onto your computer.

Windows

```
py -3 -m robotpy_installer download-python
```

Linux/macOS

```
robotpy-installer download-python
```

Once everything has downloaded, you can switch to your Robot's network, and use the following commands to install.

Windows

```
py -3 -m robotpy_installer install-python
```

Linux/macOS

```
robotpy-installer install-python
```

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

Installing RobotPy on a roboRIO

The RobotPy installer supports downloading wheels from PyPI and the RobotPy website and installing them on the roboRIO. The `download` and `install` commands behave similar to the `pip` command, including allowing use of a 'requirements.txt' file if desired.

As mentioned above, installation needs to be done in two steps (download then install). Once you are connected to the internet:

Windows

```
py -3 -m robotpy_installer download robotpy
```

Linux/macOS

```
robotpy-installer download robotpy
```

See also:

This command only downloads the core RobotPy packages. See additional details for installing *optional/vendor components*

Once everything has downloaded, you can switch to your Robot's network, and use the following commands to install.

Windows

```
py -3 -m robotpy_installer install robotpy
```

Linux/macOS

```
robotpy-installer install robotpy
```

The robotpy installer uses pip to download and install packages, so you can replace `robotpy` above with the name of a pure python package as published on PyPI.

Note: If you need Python packages that require compilation, the RobotPy project distributes some commonly used packages. See the roborio-wheels project for more details.

Upgrading RobotPy on a roboRIO

The `download` and `install` commands support some pip options, so to upgrade you can use the `-U` flag on the commands mentioned above to download the latest versions of RobotPy.

Windows

```
py -3 -m robotpy_installer download -U robotpy
```

Linux/macOS

```
robotpy-installer download -U robotpy
```

The robotpy installer can tell you what packages you have installed on a roboRIO:

Windows

```
py -3 -m robotpy_installer list
```

Linux/macOS

```
robotpy-installer list
```

1.3.4 robotpy-cscore install

Note: cscore is not installed when you `pip install robotpy`

RoboRIO installation

If you have robotpy-installer on your computer, then installing `robotpy-cscore` is very simple:

Windows

```
# While connected to the internet
py -3 -m robotpy_installer download robotpy-cscore

# While connected to the network with a RoboRIO on it
py -3 -m robotpy_installer install robotpy-cscore
```

Linux/macOS

```
# While connected to the internet
robotpy-installer download robotpy-cscore

# While connected to the network with a RoboRIO on it
robotpy-installer install robotpy-cscore
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

Non-roboRIO installation

We now distribute wheels for CSCore on pypi, so you can just use the robotpy-meta package to install it:

Windows

```
py -3 -m pip install -U robotpy[cscore]
```

Linux/macOS

```
pip3 install -U robotpy[cscore]
```

Next steps

See our *cscore documentation* for examples and deployment thoughts.

1.3.5 Package notes

Notes specific to individual packages

pyfrc install

Installing pyfrc will install all of the packages needed to help you write and test Python-based Robot code on your development computer. These tools include WPILib, pynetworktables, unit testing support, and the *robot simulator*.

It is recommended to install using the robotpy meta package:

Windows

```
py -3 -m pip install robotpy
```

Linux/macOS

```
pip3 install robotpy
```

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/macOS

```
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 macOS it requires XCode to be installed
- On Linux you will need to have python3-dev/python3-devel or a similar package installed

pynetworktables install

Warning: This page has not been updated for 2023. We recommend using pyntcore instead of pynetworktables.

pynetworktables is a python package that allows FRC teams to use Python to communicate with their robots via NetworkTables. It should work without issues on your Driver Station, on a coprocessor such as a Raspberry Pi, or anywhere else that you might install Python.

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 macOS/Linux

On a Linux or macOS 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

Getting Started

See the *NetworkTables guide* to learn more about using pynetworktables to communicate with your robot.

Commands install

The WPILib command framework is distributed separately from WPILib, and is called robotpy-commands-v2. The instructions below discuss installing the new command framework.

Setup (tests/simulator)

If you intend to use the command framework in your *robot tests* or in the simulator, you must install this package locally:

Windows

```
py -3 -m pip install -U robotpy[commands2]
```

Linux/macOS

```
pip3 install -U robotpy[commands2]
```

Setup (RoboRIO)

Even if you have robotpy-commands-v2 installed locally, you **must** install it on your robot **separately**.

Use the RobotPy installer and run the following on your computer while connected to the internet:

Windows

```
py -3 -m robotpy_installer download -U robotpy[commands2]
```

Linux/macOS

```
python3 -m robotpy_installer install robotpy[commands2]
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

robotpy-ctre install

Setup (tests/simulator)

If you intend to use robotpy-ctre in your *robot tests* or via the robot *simulator*, you must install this package locally. It is recommended to install using the robotpy meta package:

Windows

```
py -3 -m pip install -U robotpy[ctre]
```

Linux/macOS

```
pip3 install -U robotpy[ctre]
```

Setup (RoboRIO)

Even if you have robotpy-ctre installed locally, you **must** install it on your robot **separately**. See below.

Python package

You really don't want to compile this yourself, so don't download this from pypi and install it. Use the RobotPy installer and run the following on your computer while connected to the internet:

Windows

```
py -3 -m robotpy_installer download -U robotpy[ctre]
```

Linux/macOS

```
robotpy-installer download -U robotpy[ctre]
```

Then, when connected to the roborio's network, run:

Windows

```
py -3 -m robotpy_installer install robotpy[ctre]
```

Linux/macOS

```
robotpy-installer install robotpy[ctre]
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

NI Web Dashboard (optional)

CTRE Phoenix can integrate with the NI Web Dashboard on the RoboRIO. This is not required to run robotpy-ctre on the RoboRIO, but it can be a useful diagnostic tool. To install this, you will need to use the CTRE Lifeboat tool to install it separately.

Refer to the CTRE documentation for more details.

robotpy-navx install

Setup (tests/simulator)

If you intend to use robotpy-navx in your *robot tests* or via the *pyfrc simulator*, you must install this package locally. It is recommended to install using the robotpy meta package:

Windows

```
py -3 -m pip install -U robotpy[navx]
```

Linux/macOS

```
pip3 install -U robotpy[navx]
```

Setup (RoboRIO)

Even if you have robotpy-navx installed locally, you **must** install it on your robot **separately**. Use the RobotPy installer and run the following on your computer while connected to the internet:

Windows

```
py -3 -m robotpy_installer download -U robotpy[navx]
```

Linux/macOS

```
robotpy-installer download -U robotpy[navx]
```

Then, when connected to the roborio's network, run:

Windows

```
py -3 -m robotpy_installer install robotpy[navx]
```

Linux/macOS

```
robotpy-installer install robotpy[navx]
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

robotpy-pathplannerlib install

Setup (tests/simulator)

If you intend to use robotpy-pathplannerlib in your *robot tests* or via the *pyfrc simulator*, you must install this package locally. It is recommended to install using the robotpy meta package:

Windows

```
py -3 -m pip install -U robotpy[pathplannerlib]
```

Linux/macOS

```
pip3 install -U robotpy[pathplannerlib]
```

Setup (RoboRIO)

Even if you have robotpy-pathplannerlib installed locally, you **must** install it on your robot **separately**. See below.

Python package

You really don't want to compile this yourself, so don't download this from pypi and install it. Use the RobotPy installer and run the following on your computer while connected to the internet:

Windows

```
py -3 -m robotpy_installer download -U robotpy[pathplannerlib]
```

Linux/macOS

```
robotpy-installer download -U robotpy[pathplannerlib]
```

Then, when connected to the roborio's network, run:

Windows

```
py -3 -m robotpy_installer install robotpy[pathplannerlib]
```

Linux/macOS

```
robotpy-installer install robotpy[pathplannerlib]
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

robotpy-photonvision install

Setup (tests/simulator)

If you intend to use robotpy-photonvision in your *robot tests* or via the *pyfrc simulator*, you must install this package locally. It is recommended to install using the robotpy meta package:

Windows

```
py -3 -m pip install -U robotpy[photonvision]
```

Linux/macOS

```
pip3 install -U robotpy[photonvision]
```

Setup (RoboRIO)

Even if you have robotpy-photonvision installed locally, you **must** install it on your robot **separately**. See below.

Python package

You really don't want to compile this yourself, so don't download this from pypi and install it. Use the RobotPy installer and run the following on your computer while connected to the internet:

Windows

```
py -3 -m robotpy_installer download -U robotpy[photonvision]
```

Linux/macOS

```
robotpy-installer download -U robotpy[photonvision]
```

Then, when connected to the roborio's network, run:

Windows

```
py -3 -m robotpy_installer install robotpy[photonvision]
```

Linux/macOS

```
robotpy-installer install robotpy[photonvision]
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

robotpy-playingwithfusion install

Setup (tests/simulator)

If you intend to use robotpy-playingwithfusion in your *robot tests* or via the *pyfrc simulator*, you must install this package locally. It is recommended to install using the robotpy meta package:

Windows

```
py -3 -m pip install -U robotpy[playingwithfusion]
```

Linux/macOS

```
pip3 install -U robotpy[playingwithfusion]
```

Setup (RoboRIO)

Even if you have robotpy-playingwithfusion installed locally, you **must** install it on your robot **separately**. See below.

Python package

You really don't want to compile this yourself, so don't download this from pypi and install it. Use the RobotPy installer and run the following on your computer while connected to the internet:

Windows

```
py -3 -m robotpy_installer download -U robotpy[playingwithfusion]
```

Linux/macOS

```
robotpy-installer download -U robotpy[playingwithfusion]
```

Then, when connected to the roborio's network, run:

Windows

```
py -3 -m robotpy_installer install robotpy[playingwithfusion]
```

Linux/macOS

```
robotpy-installer install robotpy[playingwithfusion]
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

robotpy-rev install

Setup (tests/simulator)

If you intend to use robotpy-rev in your *robot tests* or via the *pyfrc simulator*, you must install this package locally. It is recommended to install using the robotpy meta package:

Windows

```
py -3 -m pip install -U robotpy[rev]
```

Linux/macOS

```
pip3 install -U robotpy[rev]
```

Setup (RoboRIO)

Even if you have robotpy-rev installed locally, you **must** install it on your robot **separately**. See below.

Python package

You really don't want to compile this yourself, so don't download this from pypi and install it. Use the RobotPy installer and run the following on your computer while connected to the internet:

Windows

```
py -3 -m robotpy_installer download -U robotpy[rev]
```

Linux/macOS

```
robotpy-installer download -U robotpy[rev]
```

Then, when connected to the roborio's network, run:

Windows

```
py -3 -m robotpy_installer install robotpy[rev]
```

Linux/macOS

```
robotpy-installer install robotpy[rev]
```

For additional details about running robotpy-installer on your computer, see the *robotpy-installer documentation*.

REV Firmware and Diagnostics

robotpy-rev supports all the control features of the C++ Spark Max library. Firmware, diagnostics, and other things must be installed separately using the tools released by REV.

Refer to the REV C++ documentation for more details.

RoboRIO Package Installer

Note: This is not the RobotPy installation guide, see *Robot Installation* if you're looking for that!

Most FRC robots are not placed on networks that have access to the internet, particularly at competition arenas. The RobotPy installer is designed for this type of 'two-phase' operation – with individual steps for downloading and installing packages separately.

The RobotPy installer supports downloading external packages from the python package repository (pypi) via pip, and installing those packages onto the robot. We cannot make any guarantees about the quality of external packages, so use them at your own risk.

Note: If your robot is on a network that has internet access, then you can manually install packages via opkg or pip. However, if you use the RobotPy installer to install packages, then you can easily reinstall them on your robot in the case you need to reimage it.

If you choose to install packages manually via pip, keep in mind that 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"
```

Each of the commands supports various options, which you can read about by invoking the –help command.

Installing/Executing the installer

Note: The installer is included when you install the RobotPy meta package, so if you installed that then you likely already have it

To install/use the installer, you must have Python 3.6+ installed. You should install the installer via pip.

Windows

```
py -3 -m pip install robotpy-installer
```

Linux/macOS

```
pip3 install robotpy-installer
```

To upgrade the installed version of the installer, you need to add the `-U` flag to pip.

Executing the installer

Once you have the installer program installed, to execute the installer do:

Windows

```
py -3 -m robotpy_installer [command..]
```

Linux/macOS

```
robotpy-installer [command..]
```

Python

These commands allow you to install/upgrade Python on your roboRIO. Once Python is installed, it's likely that you won't need to upgrade it.

download-python

Windows

```
py -3 -m robotpy_installer download-python
```

Linux/macOS

```
robotpy-installer download-python
```

This will update the cached Python package to the newest versions available.

install-python

Windows

```
py -3 -m robotpy_installer install-python
```

Linux/macOS

```
robotpy-installer install-python
```

Note: You must already have Python downloaded (via `download-python`), or this command will fail.

Python Packages

If you want to use a python package hosted on Pypi in your robot code, these commands allow you to easily download and install those packages.

Note: If you need Python packages that require compilation, the RobotPy project distributes some commonly used packages. See the roborio-wheels project for more details.

---

download

Windows

```
py -3 -m robotpy_installer download PACKAGE [PACKAGE ..]
```

Linux/macOS

```
robotpy-installer download PACKAGE [PACKAGE ..]
```

Specify python package(s) to download, similar to what you would pass the 'pip install' command. This command does not install files on the robot, and must be executed from a computer with internet access.

You can run this command multiple times, and files will not be removed from the download cache.

You can also use a *requirements.txt* file to specify which packages should be downloaded.

Windows

```
py -3 -m robotpy_installer download -r requirements.txt
```

Linux/macOS

```
robotpy-installer download -r requirements.txt
```

install

Windows

```
py -3 -m robotpy_installer install PACKAGE [PACKAGE ..]
```

Linux/macOS

```
robotpy-installer install PACKAGE [PACKAGE ..]
```

Copies python packages over to the roboRIO, and installs them. If the package already has been installed, it will be reinstalled.

You can also use a *requirements.txt* file to specify which packages should be downloaded.

Windows

```
py -3 -m robotpy_installer install -r requirements.txt
```

Linux/macOS

```
robotpy-installer install -r requirements.txt
```

Warning: The 'install' command will only install packages that have been downloaded using the 'download' command, or packages that are on the robot's pypi cache.

Warning: If your robot does not have a python3 interpeter installed, this command will fail. Run the *install-python* command first.

1.4 Programmer's Guide

- If you don't know python very well (or at all), start with *Introduction to Python*
- Otherwise, start with *Anatomy of a robot*

1.4.1 Introduction to Python

Note: This is intended to be a *very* brief overview/reference of the various topics you need to master in order to program Python. This is not an exhaustive guide to programming with python. We recommend other resources to really learn that in-depth:

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

If you want to practice some of these concepts, try out pybasictraining!

Language elements

Comments

Comments are not functional and do not do anything. They are intended to be human readable things that help others understand what the code is doing. Comments should be indented at the same level as surrounding code

```
# This is a comment. It starts with a '#' character
```

Indentation

All code should be indented in multiples of 4 spaces. Tab characters should be avoided. Anytime you see a `:` character at the end of a line, the next line(s) should have another level of indentation. This is a visual indicator to show that the following lines are part of a block of code associated with the previous line. For example:

```
# this is good
if x == True:
    do_something()

# this is bad and will not work
if x == True:
do_something()
```

Pass

pass is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed. **Most finished code will not use pass for anything.**

```
if False:
    pass
```

Numbers

You can use numbers in Python, and perform computations on them.

```
# A number
1

# Multiplying two numbers
2*2
```

Strings

Strings are groups of words that you can use as variables, and the program can manipulate them and pass them around.

```
"This is a string"
'This is also a string'
'''This is a string that can
   be extended to multiple lines'''
"""This is also
   a multiline string"""
```

Booleans

Boolean values are those that are True or False. In python, True and False always have the first letter capitalized.

Variables

Variables are used to store some information (strings, numbers, objects, etc). They can be assigned values, and referred to later on.

```
x = 1
x = 'some value'
```

Control Flow

If

If statements allow you to control the flow of the program and make decisions about what the program should do next, based on information retrieved previously. Note that the body of the if statement is indented:

```
if statement is True:
    # do_something here
elif other_statement is True:
    # do something lese here
else:
    # otherwise do this
```

Also see the python documentation.

Operations

Python supports various types of operations on variables and constants:

```
# Addition
1 + 2
x + 1

# Multiplication
1 * 2
```

(continues on next page)

(continued from previous page)

```
x * 2

# Equality operator (evaluated to True or False)
1 == 1
x == 1

# Less than
x < 2

# Lots more!
```

Functions

Functions are blocks of code that can be reused and are useful for grouping code. They can return a value to the caller. The code that is contained inside of a function is not executed until you call the function.

Defintion

To define a function, you start with the word `def`, followed by the name of the function, and a set of parentheses:

```
def function_name():
    '''String that describes what the function does'''
    pass
```

Functions can accept input from their callers, allowing them to be reused for many purposes. You place the names of the parameters inside the parentheses:

```
def function_name(parameter1, parameter2):
    pass
```

After computing a result, you can return it to the caller. You can also return constants or variables:

```
def function_returns_computation(parameter1, parameter2):
    return parameter1 + parameter2

def function_returns_a_variable():
    x = 1
    return x

def function_returns_a_value():
    return True
```

Calling a function

The code that is contained inside of a function is not executed until you call the function. You call it by specifying the name of the function, followed by parentheses:

```
# Calling a function that takes no parameters
function_name()
```

If you wish to pass data to the function, you place the variable names (or constants) inside of the parentheses:

```
# Calling a function with two constant parameters
return_value = function_name(1, 2)

# Calling a function with two variables
return_value = function_name(x, y)
```

Classes

A collection of functions (also called methods) and variables can be put into a logical group called a 'class'.

Definition

A class named `Foo`:

```
class Foo(object):
    '''String that describes the class'''

    def __init__(self):
        '''Constructor -- this function gets called when an instance is created'''

        # store a variable in the class for use later
        self.variable = 1

    def method(self, parameter1, optional_parameter=None):
        '''A function that belongs to the Foo class. It takes
           two arguments, but you can specify only one if you desire'''
        pass
```

A class named `Bar`

```
class Bar(Foo):
    '''This class inherits from the Foo class, so anything in
       Foo is transfered (and accessible) here'''

    def __init__(self, parameter1):
        pass
```

Creating an instance

To actually use a class, you must create an instance of the class. Each instance of a class is unique, and usually operations on the class instances do not affect other instances of the same class.

```
foo = Foo()

# These are two separate instances of the Bar class, and operations on one
# do not affect the other
bar1 = Bar(1)
bar2 = Bar(1)
```

Accessing variables stored in a class instance

```
x = Foo()           # creates an instance of Foo
y = x.variable      # get the value from the instance
x.variable = 1      # set the value in the instance
```

Calling functions (methods) on a class instance

```
x = Foo()      # this creates an instance of Foo
x.method()     # this calls the function
```

Loops

for

```
for i in a_list_of_things:
    print(i)
```

while

```
while statement is True:
    do_this_until_statement_is_not_true()
```

Exceptions

```
try:
    do_something_that_might_cause_an_exception()
    if bad_thing is True:
        raise SomeException()
except ExceptionType as e:
    # this code is only executed if an ExceptionType exception is raised
    print("Error: " + e)
```

(continues on next page)

(continued from previous page)

```
finally:
    # This is always executed
    clean_up()

try:
    import wpilib
except ImportError:
    import fake_wpilib as wpilib
```

Future topics

- Lists, dictionaries, tuples
- Scope

Next Steps

Learn about the basic structure of Robot code at *Anatomy of a robot*.

1.4.2 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. See our *python primer* for a brief introduction to the python programming language.

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 `wpilib.IterativeRobot` or `wpilib.TimedRobot`. These classes define a number of functions that you need to override, which get called at various times.

- `wpilib.IterativeRobot` functions
- `wpilib.TimedRobot` functions

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

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

```
class MyRobot(wpilib.TimedRobot):

    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 Package) 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 Package) 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, mecanum, kiwi), you'll want to use the various included class to control the motors instead of writing your own code to do it. For most standard drivetrains, you'll want to use one of three classes:

- `wpilib.drive.DifferentialDrive` for differential drive/skid-steer drive platforms such as 2 or 4 wheel platforms, the Kit of Parts drive base, "tank drive", or West Coast Drive.
- `wpilib.drive.KilloughDrive` for Killough (Kiwi) triangular drive platforms.
- `wpilib.drive.MecanumDrive` for mecanum drive platforms.

For example, when you create a `wpilib.drive.DifferentialDrive` object, you can pass in motor controller instances:

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

Or you can pass in motor controller groups to use more than one controller per side:

```
self.frontLeft = wpilib.Spark(1)
self.rearLeft = wpilib.Spark(2)
self.left = wpilib.SpeedControllerGroup(self.frontLeft, self.rearLeft)

self.frontRight = wpilib.Spark(3)
self.rearRight = wpilib.Spark(4)
self.right = wpilib.SpeedControllerGroup(self.frontRight, self.rearRight)

self.drive = wpilib.drive.DifferentialDrive(self.left, self.right)
```

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.drive Package, and the FIRST WPILib Programming Guide.

Robot Operating Modes (TimedRobot)

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 TimedRobot 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 `time.sleep()` 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
"""
```

(continues on next page)

(continued from previous page)

```
import wpilib
import wpilib.drive


class MyRobot(wpilib.TimedRobot):

    def robotInit(self):
        """
        This function is called upon program startup and
        should be used for any initialization code.
        """
        self.left_motor = wpilib.Spark(0)
        self.right_motor = wpilib.Spark(1)
        self.drive = wpilib.drive.DifferentialDrive(self.left_motor, self.right_motor)
        self.stick = wpilib.Joystick(1)
        self.timer = wpilib.Timer()

    def autonomousInit(self):
        """This function is run once each time the robot enters autonomous mode."""
        self.timer.reset()
        self.timer.start()

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

        # Drive for two seconds
        if self.timer.get() < 2.0:
            self.drive.arcadeDrive(-0.5, 0)  # Drive forwards at half speed
        else:
            self.drive.arcadeDrive(0, 0)  # Stop robot

    def teleopPeriodic(self):
        """This function is called periodically during operator control."""
        self.drive.arcadeDrive(self.stick.getY(), self.stick.getX())


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.

See also:

RobotPy comes with various frameworks that make it easier to create your robot code. See the page on *Robot Code Frameworks*.

Next Steps

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

1.4.3 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/macOS

On Linux/macOS, 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
```

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/macOS

```
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*

Note: If you're just starting out with RobotPy, you'll probably find it faster (and more instructive) to start playing with your code in the simulator before you actually deploy it to a robot.

1.4.4 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/macOS

```
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/macOS

```
python3 robot.py deploy --nc
```

Note: This requires the driver station software to be connected to your robot

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/macOS

```
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 *Deploy Artifacts*.

Next Steps

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

1.4.5 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. From 2014-2019, RobotPy provided its own simulation GUI. Starting in 2020, RobotPy uses the WPILib simulation GUI instead.

Running the simulator

To run the GUI simulator, run your robot.py with the following arguments:

Windows

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

Linux/macOS

```
python3 robot.py sim
```

User interface

See the WPILib Simulation User Interface documentation for more details.

2D Field Widget and Physics

The WPILib Simulation GUI has a 2D field available, just like the original PyFRC simulator had. This allows you to drive your robot around on a virtual field – in particular, it's very useful for testing the logic of autonomous mode movements.

Note: To enable the field view, go to the 'Window' menu, and select 2D field view.

For the robot to move across the field, 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.

We have a variety of examples and documentation available:

- PyFRC API docs
- wpilib.simulation Package
- RobotPy Examples Repository

Communicating via NetworkTables

The simulator launches a NetworkTables server (just as the robot does), so it can be communicated with via standard NetworkTables tools (such as OutlineViewer, Shuffleboard, or SmartDashboard).

For this to work, you need to tell the client to connect to the IP address that your simulator is listening on (this will be `localhost` or `127.0.0.1`).

pynetworktables2js

pynetworktables2js will automatically connect to `localhost` if no arguments are given.

OutlineViewer

You can type an address in when OutlineViewer launches, then tell it to start in client mode.

Shuffleboard

Shuffleboard can be configured to connect to localhost in the preferences.

SmartDashboard

Using SmartDashboard, you need to launch the jar using the following command:

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

Next Steps

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

1.4.6 Unit testing robot code

> **Warning:** Testing is currently still broken, but expected to be available later in the 2022 season

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/macOS

```
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/macOS

```
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/macOS

```
python3 robot.py coverage test
```

Or to run coverage over the simulator:

Windows

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

Linux/macOS

```
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.

1.4.7 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:
```

(continues on next page)

(continued from previous page)

```
        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*

1.4.8 Using NetworkTables

Warning: This documentation has not been documented for the 2023 season yet

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 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.

Warning: NetworkTables does not connect instantly! If you write a script that calls `initialize` and immediately tries to read from NetworkTables, you will almost certainly not be able to read any data.

To write a script that waits for the connection, you can use the following code:

```
import threading
from networktables import NetworkTables

cond = threading.Condition()
notified = [False]

def connectionListener(connected, info):
    print(info, '; Connected=%s' % connected)
    with cond:
        notified[0] = True
        cond.notify()

NetworkTables.initialize(server='10.xx.xx.2')
NetworkTables.addConnectionListener(connectionListener, immediateNotify=True)
```

```
with cond:
    print("Waiting")
    if not notified[0]:
        cond.wait()

# Insert your processing code here
print("Connected!")
```

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

```
table = NetworkTablesInstance.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

See also:

NTCore 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.

- Download OutlineViewer

WPILib's Shuffleboard (requires Java) is the new (as of 2018) tool to replace SmartDashboard for creating custom NetworkTables-enabled dashboards.

- Download Shuffleboard

WPILib's SmartDashboard (requires Java) is an older tool used by teams to connect to NetworkTables and used as a dashboard.

- Download SmartDashboard

1.4.9 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
- RobotPy CSCore examples

1.5 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

1.5.1 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 available in RobotPy.

Note: Unfortunately, nobody has written any Python specific documentation for the new command framework. Please refer to the WPILib documentation for now.

If you're interested in contributing a Python-specific guide to the command framework, we'd love your help!

See also:

Commands V2 API

See also:

MagicBot Framework

1.5.2 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, and has been used by championship caliber teams to power their robots.

While MagicBot will tend to be more useful for complex multi-module programs, it does remove some of the boilerplate 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 `TimedRobot`:

```
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 annotating the variable names and types in your MyRobot class.

```
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):
    elevator: Elevator

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

(continues on next page)

(continued from previous page)

```
class Elevator:
    elevator_motor: wpilib.Talon

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

As you may be able to infer, by declaring in your `Elevator` class an annotation that matches an attribute in your Robot class, Magicbot automatically notices this and adds an attribute in your component with the instance as defined in your robot class.

Sometimes, it's useful to use multiple instances of the same class. You can inject into unique instances by prefixing variable names with the component variable name:

```
class MyRobot(MagicRobot):
    front_swerve: SwerveModule
    back_swerve: SwerveModule

    def createObjects(self):
        # this is injected into the front_swerve instance of SwerveModule as 'motor'
        self.front_swerve_motor = wpilib.Talon(1)

        # this is injected into the back_swerve instance of SwerveModule as 'motor'
        self.back_swerve_motor = wpilib.Talon(2)


class SwerveModule:
    motor: wpilib.Talon
```

One problem that sometimes comes up is your component may require a lot of configuration parameters. Remember, anything can be injected: integers, numbers, lists, tuples.... one suggestion for dealing with this problem is use a `namedtuple` to store your variables (note that attributes of `namedtuple` are readonly):

```
from collections import namedtuple

ShooterConfig = namedtuple("ShooterConfig", ["param1", "param2", "param3"])


class MyRobot(MagicRobot):

    shooter: Shooter
    shooter_cfg = ShooterConfig(param1=1, param2=2, param3=3)


class Shooter:
    cfg: ShooterConfig

    def execute(self):
        # you can access self.cfg.param1, self.cfg.param2, etc...
        ...
```

Variable injection in magicbot is one of its most useful features, take advantage of it in creative ways!

Note: Some limitations to notice:

- You cannot access components from the `createObjects` function
- You cannot access injected variables from component constructors. If you need to do this, define a `setup` method for your component instead, and it will be called after variables have been injected.

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. Each class that is in the python module will be inspected, and added as an autonomous mode if it has a class attribute named `MODE_NAME`.

Autonomous mode objects must implement the following functions:

- `on_enable` - Called when autonomous mode is initially enabled
- `on_disable` - Called when autonomous mode is no longer active
- `on_iteration` - Called for each iteration of the autonomous control loop

Your autonomous object may have the following attributes:

- `MODE_NAME` - The name of the autonomous mode to display to users (required)
- `DISABLED` - If True, don't allow this mode to be selected
- `DEFAULT` - If True, this is the default autonomous mode selected

You cannot access injected variables from component constructors. If you need to do so you can implement a `setup` function, which will be called after variables have been injected.

If you build your autonomous mode using the `AutonomousStateMachine` class, it makes it easier to build more expressive autonomous modes that are easier to reason about.

Here's an example autonomous mode that drives straight for 3 seconds.

```
from magicbot import AutonomousStateMachine, timed_state, state
import wpilib

# this is one of your components
from components.drivetrain import DriveTrain


class DriveForward(AutonomousStateMachine):

    MODE_NAME = "Drive Forward"
    DEFAULT = True

    # Injected from the definition in robot.py
    drivetrain: DriveTrain

    @timed_state(duration=3, first=True)
    def drive_forward(self):
        self.drivetrain.move(-0.7, 0)
```

Note that the `AutonomousStateMachine` object already defines default `on_enable`/`on_disable`/`on_iteration` methods that do the right thing.

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:
```

(continues on next page)

(continued from previous page)

```
    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
- Shuffleboard 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

1.6 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.

- Hardware APIs
- 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*).

1.7 Camera & Vision

The RobotPy project provides robotpy-cscore, which are python bindings for cscore, a high performance camera access and streaming library introduced by FIRST in 2017. It can be used to:

- Stream a USB/HTTP camera to SmartDashboard or the LabVIEW dashboard via HTTP
- Capture images from USB or HTTP camera, modify them using OpenCV/Numpy, and send them via HTTP to SmartDashboard, the LabVIEW dashboard, or a web browser.

`robotpy-cscore` is intended to be usable on any platform supported by OpenCV and Numpy, and is a more flexible and powerful alternative to solutions such as mjpg-streamer.

Note: cscore is potentially useful outside of the FIRST Robotics Competition, as it has very high performance and ease of use compared to other solutions.

1.7.1 On the RoboRIO

Warning: Image processing is a CPU intensive task, and because of the Python Global Interpreter Lock (GIL) **we do NOT recommend using robotpy-cscore directly in your robot process**. Don't do it. Really.

Instead, we provide easy to use ways to launch your camera/image processing code from your Python robot code, and it won't break simulation either! See below for details.

For more information on the GIL and its effects, you may wish to read the following resources:

- Python Wiki: Global Interpreter Lock
- Efficiently Exploiting Multiple Cores with Python

Note: The following assumes you're writing your robot code and your image processing code using RobotPy. However, if you're writing your Robot code using Java, we do have an example which would allow you to launch Python image processing code from your Java Robot code. See this file for details.

Installation

`robotpy-cscore` can be easily installed with the RobotPy installer. See *these instructions* for details.

Automatic camera streaming

If you do not wish to modify or process the images from your camera, and *only* wish to stream a single camera via HTTP to a dashboard, then you only need to add the following to your `robotInit` function:

```
wpilib.CameraServer.launch()
```

That's it! You should be able to connect to the camera using SmartDashboard, the default LabVIEW Dashboard, or if you point your browser at `http://roborio-XXXX-frc.local:1181`.

The quick vision example can be found in the RobotPy examples repository.

Image processing

Because the GIL exists (*see above*), RobotPy's WPILib implementation provides a way to run your image processing code in a separate process. This introduces a number of rules that your image processing code must follow to efficiently and safely run on the RoboRIO:

- Your image processing code must be in its own file
- Never import the `cscore` package from your robot code, it will just waste memory
- Never import the `wpilib` or `hal` packages from your image processing code

> **Warning:** `wpilib` may not be imported from two programs on the RoboRIO. If this happens, the second program will attempt to kill the first program.

vision file

The first step you need to do is create a file – let's call it `vision.py`, and stick it in the same directory as your `robot.py` file. You can also put it in a subdirectory underneath your robot code, and the robot deploy command will copy it to the robot.

See also:

Custom Image processing

robot.py

Once you have written your cscore code, in the `robotInit` function in your `robot.py` file you need to add the following line:

```
wpilib.CameraServer.launch('vision.py:main')
```

The parameter provided to launch is of the form `FILENAME:FUNCTION`. For example, if your code was located in the `camera` subdirectory in a file called `targeting.py`, and your function was called `run`, then you would do:

```
wpilib.CameraServer.launch('camera/targeting.py:run')
```

Important notes

- Your image processing code will be launched via a stub that will setup logging and initialize pynetworktables to talk to your robot code
- The child process will NOT be launched when running the robot code in simulation or unit testing mode
- If your image processing code contains a `if __name__ == '__main__':` block, the code inside that block will NOT be executed when the code is launched from `robot.py`
- The camera code will be killed when the `robot.py` program exits. If you wish to perform cleanup, you should register an atexit handler.

The intermediate vision example can be found in the RobotPy examples repository.

More information

- The WPILib documentation for cscore may be useful to explain concepts (though some details are different)
- *CSCore Troubleshooting*

1.7.2 Other platforms

`robotpy-cscore` is a great solution for running image processing on a coprocessor such as the Raspberry Pi or on the Driver Station. However, we do not provide precompiled packages at this time and you will need to compile binary packages for your platform.

Installation

See *installing robotpy-cscore*.

Automatic camera streaming

If you do not wish to modify or process the images from your camera, and *only* wish to stream a single camera via HTTP to a dashboard, then you can use the cscore `__main__` module to start a CameraServer automatically:

```
$ python3 -m cscore
```

That's it! You can point your browser to that host at port 1181, and you should see the cscore default webpage.

Running a custom cscore program

See also:

Custom Image processing

The easiest way to launch a cscore program is via the cscore `__main__` module helper. It will automatically configure python logging for you, and also provides useful exception handling and NetworkTables configuration for you automatically.

You can instruct the `__main__` module to run your custom code via a command line parameter. The parameter is of the form `FILENAME:FUNCTION`. For example, if your code was located in a file called `targeting.py`, and your function was called `run`, then you would do:

```
$ python3 -m cscore targeting.py:run
```

Examples

See the *intermediate_cameraserver.py* example in the robotpy-cscore examples folder

Launching your script at startup

TODO: Add section about launching your script at coprocessor startup

Viewing streams via the LabVIEW dashboard or Shuffleboard

The LabVIEW dashboard and Shuffleboard both retrieve information about the camera stream via NetworkTables. If you use the `cscore.CameraServer` class to manage your streams (which you should!) it will automatically publish the correct information to NetworkTables. In order for the dashboard program to receive the NetworkTables information, you need to tell your cscore program to connect to a NetworkTables server (your robot), and the robot code must be actually running.

If you're using the cscore `__main__` module to launch your code you can tell it configure NetworkTables to connect to your robot. Use the `--robot` or `--team` options:

```
$ python3 -m cscore --team XXXX
```

Or if running custom code:

```
$ python3 -m cscore --team XXXX vision.py:run
```

If you're writing your own custom code/launcher, at some point you should initialize NetworkTables and point it at your robot:

```
import networktables
networktables.startClientTeam(1234)
```

More information

- The WPILib documentation for cscore may be useful to explain concepts (though some details are different)
- *CSCore Troubleshooting*

1.7.3 Custom Image processing

vision file

Warning: If you merely wish to display a single camera stream and do not want to process the images, do **NOT** use this code. Instead, see one of the following sections about automatic streaming:

- *RoboRIO automatic streaming*
- *non-RoboRIO automatic streaming*

The first step you need to do is create a file – let's call it `vision.py`. `vision.py` must contain some function to be called, let's call it `main`, and at the minimum it needs to do the following operations:

- Create a CameraServer instance
- Start capturing from USB
- Get a cvSink object that images can be retrieved from
- Loop and capture images

Here's a full example:

```
# Import the camera server
from cscore import CameraServer

# Import OpenCV and NumPy
import cv2
import numpy as np

def main():
    cs = CameraServer.getInstance()
    cs.enableLogging()

    # Capture from the first USB Camera on the system
    camera = cs.startAutomaticCapture()
    camera.setResolution(320, 240)

    # Get a CvSink. This will capture images from the camera
    cvSink = cs.getVideo()

    # (optional) Setup a CvSource. This will send images back to the Dashboard
    outputStream = cs.putVideo("Name", 320, 240)

    # Allocating new images is very expensive, always try to preallocate
    img = np.zeros(shape=(240, 320, 3), dtype=np.uint8)

    while True:
        # Tell the CvSink to grab a frame from the camera and put it
        # in the source image.  If there is an error notify the output.
        time, img = cvSink.grabFrame(img)
        if time == 0:
            # Send the output the error.
            outputStream.notifyError(cvSink.getError());
            # skip the rest of the current iteration
            continue

        #
        # Insert your image processing logic here!
        #

        # (optional) send some image back to the dashboard
        outputStream.putFrame(img)
```

This code will work both on a RoboRIO and on other platforms. The exact mechanism to run it differs depending on whether you're on a RoboRIO or a coprocessor:

- *RoboRIO*
- *Other*

Multiple Cameras

cscore easily supports multiple cameras! Here's a really simple `vision.py` file that will get you started streaming two cameras to the FRC Dashboard program:

```
from cscore import CameraServer

def main():
    cs = CameraServer.getInstance()
    cs.enableLogging()

    usb1 = cs.startAutomaticCapture(dev=0)
    usb2 = cs.startAutomaticCapture(dev=1)

    cs.waitForever()
```

One thing to be careful of: if you get USB Bandwidth errors, then you probably need to do one of the following:

- Reduce framerate (FPS). The default is 30, but you can get by with 10 or even as low as 5 FPS.
- Lower image resolution: you'd be surprised how much you can do with a 160x120 image!

Sometimes the first and second camera swap!?

When using multiple USB cameras, Linux will sometimes order the cameras unpredictably – so camera 1 will become camera 0. Sometimes.

The way to deal with this is to tell cscore to use a specific camera by its path on the file system. First, identify the cameras `dev` paths by using SSH to access the robot and execute `find /dev/v4l`. You should see output similar to this:

```
/dev/v4l
/dev/v4l/by-path
/dev/v4l/by-path/pci-0000:00:1a.0-usb-0:1.4:1.0-video-index0
/dev/v4l/by-path/pci-0000:00:1d.0-usb-0:1.4:1.2-video-index0
/dev/v4l/by-id
...
```

What you need to do is figure out what paths belong to which camera, and then when you start the camera server, pass it a name and a path via:

```
usb1 = cs.startAutomaticCapture(name="cam1", path='/dev/v4l/by-id/some-path-here')
usb2 = cs.startAutomaticCapture(name="cam2", path='/dev/v4l/by-id/some-other-path-here')
```

Generally speaking, if your cameras have unique IDs associated with them (you can tell because the by-id path has a random string of characters in it), then using by-id paths are the best, as they'll always be the same regardless which port the camera is plugged into.

However, if your camera does NOT have unique IDs associated with them, then you should use the by-path versions instead. These device paths are unique to each USB port plugged in. They should be fairly deterministic, but sometimes with USB hubs they have been known to change.

Note: The Microsoft Lifecam cameras commonly used in FRC don't have unique IDs associated with them, so you'll want to use the `by-path` versions of the links if you are using two Lifecams.

More information

- The WPILib documentation for cscore may be useful to explain concepts (though some details are different)
- robotpy-cscore API documentation
- *CSCore Troubleshooting*

1.8 Troubleshooting

1.8.1 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 WPILib 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/macOS

```
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 robotpy via pip*?

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

Problem: no module named ...

If you're using a non-WPILib vendor library, it must be installed separately.

- *robotpy-ctre install*
- *robotpy-navx install*
- *robotpy-photonvision install*
- *robotpy-playingwithfusion install*
- *robotpy-rev install*

If you're on your local computer, did you *install robotpy 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/macOS. 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 robotpy installation on your computer to match the version on the robot. Just run:

Windows

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

Linux/macOS

```
pip3 install --upgrade robotpy
```

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.

Problem: My code segfaulted and there's no Python stack trace!

When you find something like this here's what you can do:

First, figure out where the code is crashing. Traditional debugging techniques apply here, but a simple way is to just delete and/or comment out things until it no longer fails. Then add the last thing back in and verify that the code still crashes.

Advanced users can compile a custom version of the robotpy libraries with symbols and use gdb to get a full stack trace (documentation TBD).

Once you've identified where it crashes, file a bug on github and we can help you out.

Common causes

Python objects are reference counted, and sometimes when you pass one directly to a C++ function without retaining a reference a crash can occur:

```
class Foo:
    def do_something(self):
        some_function(Thing())
```

In this example, `Thing` is immediately destroyed after some_function returns (because there are no references to it), but some_function (or something else) tries to use the object after it is destroyed. This causes a segfault or memory access exception of some kind.

These are considered bugs in RobotPy code and if you report an issue on github we can fix it. However, as a workaround you can retain a reference to the thing that you created and that often resolves the issue:

```
class Foo:
    def do_something(self):
        self.thing = Thing()
        some_function(self.thing)
```

1.8.2 pynetworktables

isConnected() returns False!

Keep in mind that NetworkTables does not immediately connect, and it will connect/disconnect as devices come up and down. For example, if your program initializes NetworkTables, sends a value, and exits – that almost certainly will fail.

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

# replace your team number below
NetworkTables.startClientTeam(1234)
```

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`.

1.8.3 cscore

Problem: I can't view my cscore stream via a dashboard

First, make sure that your stream is actually working. Connect with a web browser to the host that the stream is running on on the correct port (if you are using CameraServer, this will be output via a python logging message). The default port is 1181.

The LabVIEW dashboard and Shuffleboard both receive information about connecting to the stream via NetworkTables. This means that both your cscore code and the dashboard need to be connected to your robot, and your robot's code needs to be running. If you have python logging enabled, then your cscore code should output a message like this if it's connected to a robot:

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

If it's connected to NetworkTables, then you can use something like the TableViewer to view the contents of NetworkTables and see if the correct URL is being published. Look under the 'CameraPublisher' key.

Problem: My image processing code is running at 100% CPU usage

You should only encounter this if running your own image processing code. If you're just streaming a camera, this should never happen and is a bug. When doing image processing, there's a few ways you can use too much CPU, particularly if you do it on a RoboRIO. Here are some thoughts:

- Resizing images is really expensive, don't do that. Instead, set the resolution of your camera via the API provided by cscore
- Preallocate your image buffers. Most OpenCV functions will optionally take a final argument called 'dst' that it will write the result of the image processing operation to. If you don't provide a 'dst' argument, then it will allocate a new image buffer each time. Because image buffers can be really large, this adds up quickly.
- Try a really small resolution like 160x120. Most image processing tasks for FRC are still perfectly doable at small resolutions.

- If your framerate is over 10fps, consider bringing it down and see if that helps.

Problem: It still doesn't work!

Please file a bug on github or use one of our *support channels*.

1.9 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

> **Warning:** When posting on ChiefDelphi, post on the Python forum, **not** the main programming forum, otherwise your support request may get lost in the noise!

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!

1.9.1 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.

1.9.2 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.

1.9.3 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.

1.10 FAQ

Here you can find answers to some of the most frequently asked questions about RobotPy.

1.10.1 Should our team use RobotPy?

What we often recommend teams to do is to take their existing code for their existing robot, and translate it to RobotPy and try it out first in the robot simulator, followed by the real robot. This will give you a good taste for what developing code for RobotPy will be like.

Related questions for those curious about RobotPy:

- *Is RobotPy competition-legal?*
- *Is RobotPy stable?*
- *Is RobotPy fast?*

1.10.2 Installing and Running RobotPy

How do I install RobotPy?

See our *getting started guide*.

What version of Python do RobotPy projects use?

When running RobotPy on a FIRST Robot, our libraries/interpreters use Python 3. This means you should reference the Python 3.x documentation instead of the Python 2.x documentation.

- RobotPy WPILib on the roboRIO uses the latest version of Python 3 at kickoff. In 2023, this was Python 3.11. When using pyfrc or similar projects, you should use a Python 3.7 or newer interpreter (the latest is recommended).
- RobotPy 2014.x is based on Python 3.2.5.

pynetworktables is compatible with Python 3.5 or newer, since 2019. Releases prior to 2019 are also compatible with Python 2.7.

What happens when my code crashes?

An exception will be printed out to the console, and the Driver Station log may receive a message as well. It is highly recommended that you enable NetConsole for your robot, so you can see these messages.

Is WPILib available?

Of course! Just `import wpilib`. Class and function names are identical to the Java version. Check out the Python WPILib API Reference for more details.

As of 2020, the API mostly matches the C++ version of WPILib, except that protected functions are prefixed with an underscore (but are availble to all Python code).

From 2015-2019, almost all classes and functions from the Java WPILib are available in RobotPy's WPILib implementation.

Prior to 2015, the API matched the C++ version of WPILib.

Is Command-based programming available?

Of course! Check out the `commands2` package. There is also some *python-specific documentation available*.

Is there an easy way to test my code outside of the robot?

Glad you asked! Our pyfrc project has a built in *lightweight robot simulator* you can use to run your code, and also has builtin support for unit testing with py.test.

1.10.3 Competition

Is RobotPy competition-legal?

Python is not an official FRC language yet, but we are working with the WPILib team to make it official in 2024. See https://wpilib.org/blog/bringing-python-to-frc for the announcement.

As RobotPy was not written by anyone involved with the GDC, we can't provide a guaranteed answer (particularly not for future years). However, we see no reason that RobotPy would not be legal: to the cRIO/RoboRIO, it looks just like any other C++ WPILib-using program that reads text files. RobotPy itself should be considered COTS software as it is freely available to all teams. Teams have been using RobotPy since 2010 without any problems from FIRST, and we expect that to continue.

Caveat emptor: while RobotPy is almost certainly legal to use, your team should carefully consider the risk of using such a large piece of unofficial software; unless RobotPy is used by many teams, if you run into trouble at a competition, there may not be anyone else there to help! However, we've found that most problems teams run into are problems with WPILib itself, and not RobotPy.

Also, be sure to keep in mind the fact that Python is a dynamic language and is NOT compiled. This means that typos can easily go undetected until your robot runs that particular line of code, resulting in an exception and 5 second restart. Make sure to test your code thoroughly (see our *unit testing documentation*).

Is RobotPy stable?

Yes! While Python is not an officially supported language, teams have been using RobotPy since 2010, and the maintainer of RobotPy is a member of the WPILib team. Much of the time when bugs are found, they are found in the underlying WPILib, instead of RobotPy itself.

One caveat to this is that because RobotPy doesn't have a beta period like WPILib does, bugs tend to be found during the first half of competition season. However, by the time build season ends, RobotPy is just as stable as any of the officially suported languages.

How often does RobotPy get updated?

RobotPy is a community project, and updates are made whenever community members contribute changes and the developers decide to push a new release.

Historically, RobotPy tends to have frequent releases at the beginning of build season, with less frequent releases as build season goes on. We try hard to avoid WPILib releases after build season ends, unless critical bugs are found.

1.10.4 Performance

Is RobotPy fast?

It's fast enough.

We've not yet benchmarked it, but it's almost certainly just as fast as Java for typical WPILib-using robot code. RobotPy uses the native C++ WPILib, and thus the only interpreted portions are your specific robot actions. If you have particularly performance sensitive code, you can write it in C++ and use pybind11 wrappers to interface to it from Python.

1.10.5 RobotPy Development

Who created RobotPy?

RobotPy was created by Peter Johnson, programming mentor for FRC Team 294, Beach Cities Robotics. He was inspired by the Lua port for the cRIO created by Ross Light, FRC Team 973. Peter is a member of the FIRST WPILib team, and also created the ntcore and cscore libraries.

The current RobotPy maintainer is Dustin Spicuzza, also a member of the FIRST WPILib team.

Current RobotPy developers include:

- Dustin Spicuzza (@virtuald)
- David Vo (@auscompgeek)
- Vasista Vovveti (@TheTripleV)

1.10.6 How can I help?

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

- Test and report any issues you find.
- Port and test a useful library.
- Write a Python module and share it with others (and contribute it to the robotpy-wpilib-utilities package!)

1.11 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. :)

1.11.1 Design

Adding options to robot.py

When `wpilib.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):
```

(continues on next page)

(continued from previous page)

```
        # 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']},
      ...
      )
```

1.11.2 Developer Installation

TODO

1.11.3 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!

Deploy Artifacts

During the deploy process, robotpy will generate a `deploy.json` that can provide your robot with extra information regarding how code was deployed. It contains information that could be used for example, to alert you if your hash contains the git `-dirty` flag, or assist in debugging an issue by exploring who, when and with what code was deployed with.

The `deploy.json` is not present on the dev filesystem and is copied over via sftp at deploy time. It contains the following keys:

```
{
 "git-desc": "2022.1-8-gb4fc2aca-dirty",
 "git-hash": "b4fc2aca399810f1fe28faf23314cd422a6db920",
 "git-branch": "feat/working_code",
 "deploy-host": "MyLaptop",
```

(continues on next page)

(continued from previous page)

```
 "deploy-user": "me",
 "deploy-date": "2018-6-10T02:40:55",
 "code-path": "/home/me/robots/MyRobotCode"
}
```

If you do not manage your code with git, use another VCS, or do not have git installed locally and on your path in the usual location, the git tag will not be present.

You can use `./robotpy.py deploy-info` to connect to the robot and fetch the deploy.json.

Example code:

```
#!/usr/bin/env python3

import os
import json
import wpilib.deployinfo


class MyRobot(wpilib.TimedRobot):
   def robotInit(self):
      data = wpilib.deployinfo.getDeployData()

      print(data)

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

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.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX