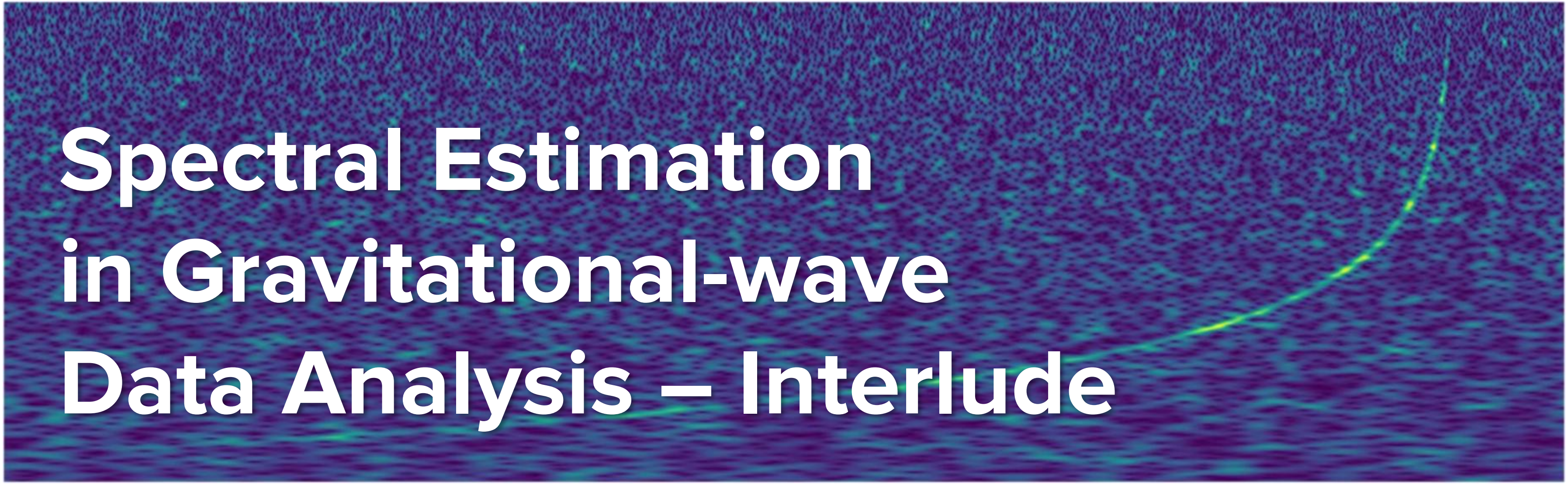


Spectral Estimation in Gravitational-wave Data Analysis – Interlude

Gravitational Wave Detector Characterization Workshop

Inter-University Centre for Astronomy and Astrophysics, Pune – December 15-19, 2025

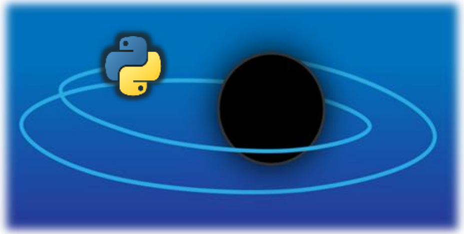
Francesco Di Renzo, University of Florence, Italy

UNIVERSITÀ DEGLI STUDI FIRENZE

LIGO VIRGO KAGRA

Python Tools Powering the GW Science Stack

- Python is currently the most widespread computing language in Gravitational Wave Astronomy, especially within the LVK
- It is the preferred language for Data Analysis pipelines
- Most tutorials, publications, and repositories (e.g., GWOSC, Zenodo, the next tutorials) are written in Python
- Public alerts and parameter estimation reports are generated using Python-based tools



Key Python Libraries:

- PyCBC: Matched filtering, SNR analysis, inspiral waveform generation
- GWpy: Time series and spectrogram analysis tailored to LIGO data
- Bilby: Bayesian inference for compact object mergers
- ligo.skymap: Skymap generation and localization
- Astropy: Collection of software packages for use in Astronomy
- Python Virgotools: Functions to interact with the Virgo interferometer’s software, hardware, and data.

Operations White Paper – OPS-8.5
Python is recommended for development to maximize compatibility with existing tools, reducing duplication-of-effort and redundancy.

Why Python has become Critical

Ecosystem Integration

- Python easily integrates with:
  - **C/C++** (via bindings like SWIG or ctypes), still vital for performance-heavy components (e.g., LALSuite core)
  - **Shell scripts** (`subprocess`) for job management and automate workflows on computing clusters
  - **Web APIs** (Application Programming Interfaces, using `requests` module)
- Used to glue together workflows on computing clusters and notebooks alike:
  - Python scripts manage **job submission**, **data transfer**, and **pipeline execution**

Rapid Prototyping and Collaboration-Friendly

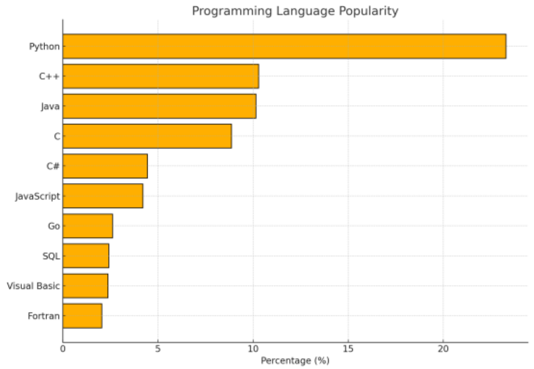
- Scientists write test scripts, visualizations, and simulations quickly with Python
- Ideal for collaborative work and reproducibility
- Rich ecosystem of open-source tools (Jupyter, GitHub, Conda, etc.)
- Huge online support community: Stack Overflow, forums, GitHub Python, etc.

Python & Object-Oriented Programming

Minimal Introduction for Gravitational-wave Enthusiasts

What is Python?

- Created in the late 1980s by Guido van Rossum (after Monty Python)
- High-level, readable language, great for beginners or as a "first language"
- Interpreted: runs line-by-line



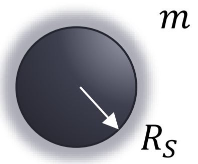
What is Object-Oriented Programming (OOP)?

- A way of thinking about programs as **collections of objects:** Objects = data + behavior
- Helps organize and reuse code like we organize real-world things:
    - **Class** – A blueprint or template (e.g., "a car")
    - **Object** – An instance of a class (e.g., "a Toyota Yaris Hybrid")
    - **Attributes** – Characteristics of a Class (e.g., "white", "115 CV")
    - **Methods** – Actions by objects (e.g., "clean", "refuel")

Python Basics with a GW Twist

Key Concepts:

- Variables store data (e.g., event name, mass)
- Functions performs actionsencapsulate physics (e.g. Schwarzschild radius)



Code Example:

```
# Variables (this is a line comment BTW!)
name = “GW150914” # string
mass = 30  # in solar masses
spin = 0.7 # dimensionless spin

# Function
def schwarzschild_radius(m):
    """Function to return the Schwarzschild
    radius of a mass expressed in Solar Masses.

    And this is a block comment BTW (docstring)"""
    return 2 * m * 1.476  # km

print(f“Primary component mass of event {name}:”)
print(schwarzschild_radius(mass))
```

OOP to Model Astrophysical Objects

Key Concepts:

- OOP is about classes (blueprints) and objects (instances): parallel with physical modeling
- **Class:** a type of thing (e.g., BlackHole)
- **Object:** an instance (e.g., bh1 = BlackHole(...))
- **Attributes:** mass, spin
- **Methods:** things it can do (e.g., merge)

Code Example:

```
# Class
class BlackHole:
    def __init__(self, mass, spin):
        self.mass = mass
        self.spin = spin

    def __str__(self):
        return f"BlackHole(mass={self.mass},
                 spin={self.spin})"
```

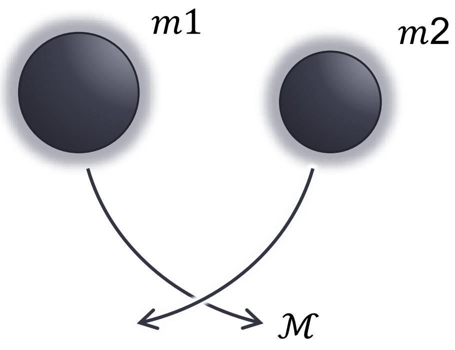
Contents:

- `__init__` creates the object
- `self.mass`, `self.spin` are attributes
- `__str__` defines how it prints

Simulating Mergers with OOP and Methods

New concepts:

- `merge()` is a method that interacts with another object
- `Coalesce` is a result class
- The *chirp mass* is calculated for these two objects coalescing



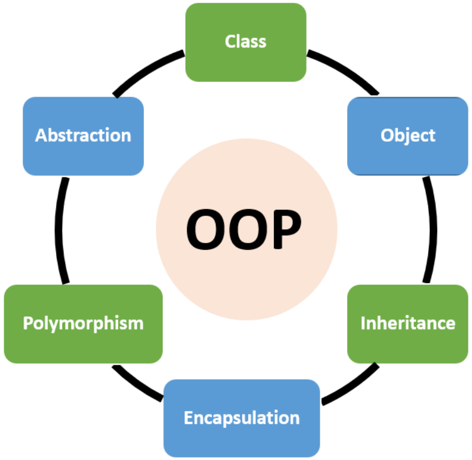
Code Example:

```
# Another class
class Coalescence:
    def __init__(self, bh1, bh2):
        self.total_mass = bh1.mass + bh2.mass
        self.chirp = self.total_mass ** (5/3)

class BlackHole:
    ...
    def merge(self, other):
        return Coalescence(self, other)

# Example:
bh1 = BlackHole(30, 0.7)
bh2 = BlackHole(25, 0.5)
gw = bh1.merge(bh2)
print(gw.chirp)
```

Why Use OOP for Science?

Benefits of OOP modeling:

- **Structure:** Keep code clean and modular
- **Reusability:** Use your BlackHole class in many contexts
- **Scalability:** Add features like position, velocity, collision_type
- **Intuition:** Think like a scientist, code like one



Extension Ideas:

- Create a `NeutronStar` class
- Add gravitational redshift method
- Simulate a population of black holes (which mass function?)
- Animate the chirp using matplotlib (PyCBC can come handy…)

What we will see next makes abundant use of OOP… Be prepared!

GWpy: Python Tools for GW Data Analysis

GWpy is Python package designed to simplify access, visualization, and manipulation of gravitational-wave time series and spectral data, developed for and by the LIGO/Virgo collaboration. It is supported by a thorough documentation and extensive list of examples available online.

Key Features:

- Read data from frame files, GWFs, or NDS servers
- Analyze time series (`TimeSeries`) and frequency series (`FrequencySeries`)
- Handles Data Quality flags (`DataQualityFlag`) and event lists, such as GW events, glitch triggers, etc. (`EventTable`)
- Extensive signal processing built-in functionalities
- Easily read open data (`TimeSeries.fetch_open_data()`)
- Generate publication-quality plots (via matplotlib): many of the plots that you see on LVK publications have been partly realized using the GWpy module

Conda Environments: Reproducible Research in a Box

What is a Conda Environment?

- A self-contained software environment, includes Python, packages, libraries, and dependencies. Think of it as a virtual lab that's isolated from the rest of your system

Why Use It?

- Avoid conflicts between different projects (e.g., Python 3.9 for one, 3.11 for another)
- Set up **exactly the right software** for your analysis (and no more) with easy version control
- Portable and cross-platform (works on Linux, macOS, Windows)
- Makes it easy to share reproducible environments among collaborators
- Easily install or roll back packages.

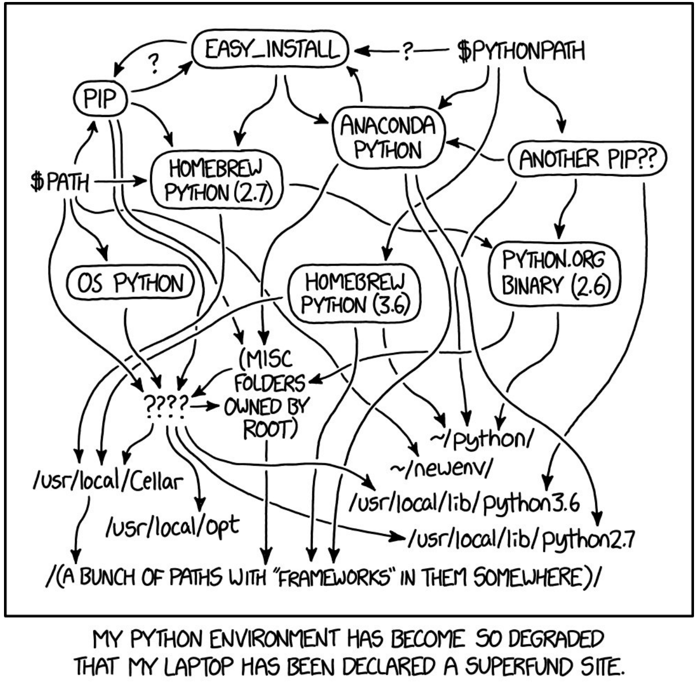


Image from XKCD Webcomic. Licensed here: https://xkcd.com/license.html.

IGWN Conda: Ready-to-Go Tools for Gravitational Wave Analysis

An IGWN Conda environment is a curated set of conda environments developed by the **International Gravitational-Wave Network (IGWN).** Already available on every IGWN computing centers.

Includes tools for LVK data analysis and software development:

- All the standard libraries, like: numpy, scipy, matplotlib, jupyter, pandas, etc.
- gwpy, ligo.skymap, pycbc, bilby, astropy, lal, gwdetchar, gcc, root, etc.
- Up-to-date and tested in LVK workflows

Installation

```
conda install -c conda-forge igwn-py311
conda activate igwn-py311
```

Export/Share Your Env:

```
conda env export > env.yaml
conda env create -f env.yaml
```

Pro tip: Pinning Environments
Use pinned environments (e.g. igwn-py39-20230425) to ensure stability and avoid surprises during paper preparation or collaboration.

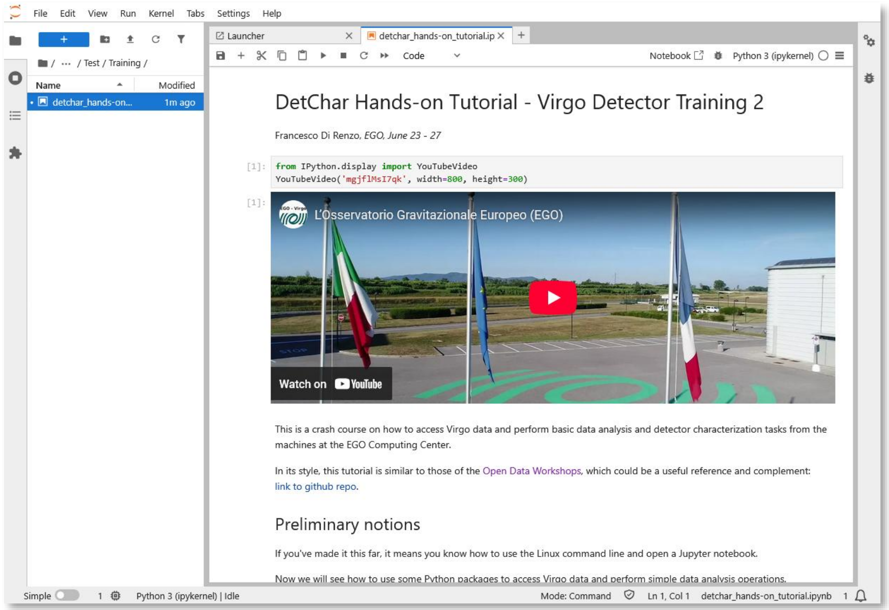
Avoid: Installing random pip packages inside Conda envs.
Keep environments small and task-specific.

Jupyter: Interactive Computing for Scientists

- Jupyter is an open-source project for interactive, literate programming
- Supports code + text + plots + outputs (images, audio, videos, etc.) in one document
- Originally from “JUlia, PYthon, and R”. Now supports 40+ language
- Gives access to computing centers via Jupyter servers, including CIT, CC-IN2P3
- Easy file transfer and directory navigation.

Ideal For:

- Developing, prototyping & debugging
- Tutorials, lectures and presentations
- Scientific reporting.



GWOSC Tutorials
On the GWOSC website, many tutorials are available as Jupyter notebooks: link.

Jupyter Notebook Essentials: Commands & Cell Magic

Jupyter notebooks are organized in In(put) and Out(put) cells:

- **Cell Types:**
  - Code: Runs Python (or other kernel)
  - Markdown: For notes, explanations, LaTeX
  - Raw: Plain text
- **Running Cells:**
  - Shift + Enter: Run current cell, go to next
  - Ctrl + Enter: Run current cell, stay
  - Alt + Enter: Run current cell, insert new cell below

Kernels are the brain behind the Notebook:

- It is the **computational engine** that executes your code
- Executes code you run in cells, stores **variables**, **imports**, **functions,** maintains **state** until restarted or shut down.

Notebook Commands (Magics):

```
%time            # Time execution of a line
%%timeit         # Run cell multiple times
                   to get avg execution time
%matplotlib inline   # Show plots inline
%load filename.py    # Load code from
                       file into a cell
%run script.py       # Run a full script
```

Shell-like Commands:

```
!ls      # List files in
           current directory
!pwd     # Show current
           working directory
!pip list # Show installed
            Python packages
```

Jupyter Keyboard Shortcuts: Command Mode vs. Edit Mode

Modes:

- Edit mode (green border):
  Press Enter to type/edit a cell
- Command mode (blue border):
  Press Esc to manage cells

Help Tips:

H	Open all shortcut keys menu
Tab	Autocomplete
Shift + Tab	Quick help on a function
?function_name	Open docstring of function

Command Mode Shortcuts:

A	Insert cell **above**
B	Insert cell **below**
D D	**Delete** cell
Z	**Undo** cell deletion
Y	Change to **code** cell
M	Change to **Markdown** cell
Shift+M	**Merge** selected cells
0 0	Restart the kernel

The End of Interlude

Backup Slides

Python VirgoTools Overview

PythonVirgoTools is a suite of Python functions and classes designed to interact with the Virgo interferometer's software, hardware, and data. Widely used in the control room, calibration, and DetChar.

Main Capabilities:

- Frame file access: read channel data (getChannel, FrameFile.get_frame)
- Interferometer control: send/receive Cm messages to other processes (cm_send, cm_start)
- DSP (digital signal processor) interface: get/set parameters of suspension DSPs
- Configuration file parsing: read .cfg files and extract structured control parameters

Python VirgoTools Modules & Use Cases

Key Modules:

- gpstime: Time conversions between GPS, UTC, and Europe/Rome local time
- frame_lib: Access and iterate Virgo frame files (.gwf, .ffl) for channel data
- cm_lib: Interface with Virgo Cm messaging system to send commands
- interface_DSP: Read and write values from suspension DSPs with smooth ramping
- cfg_lib: Extract structured info from .cfg control files
- vpm: Query VirgoProcessMonitoring XML for real-time status

Code Example: Read recent IMC transmission data

```
x = getChannel('raw', 'INJ_IMC_TRA_DC', now_gps() - 3600, 10)

# Visualize with matplotlib
plt.plot(x.time, x.data)
plt.show()
```