

Digital Communication using GNU Radio

Prof. Kumar Appaiah

Department of Electrical Engineering

Indian Institute of Technology Bombay

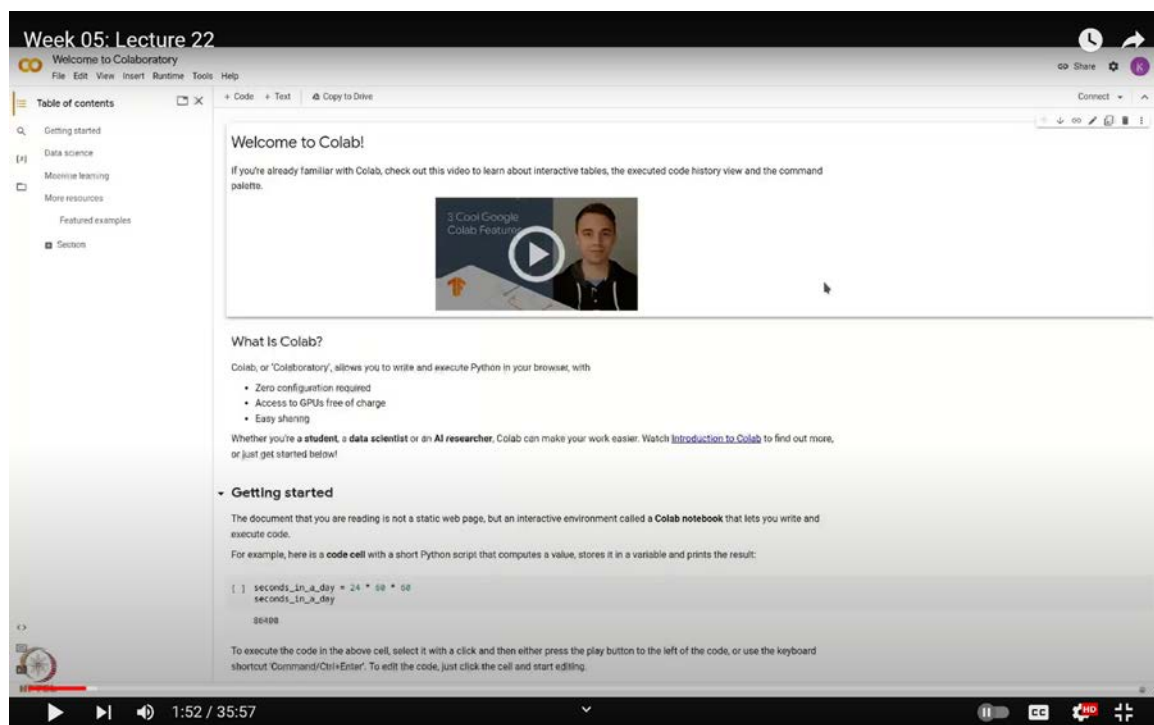
Week-05

Lecture-22

Python for GNU Radio

Welcome to this lecture on Digital Communication using GNU Radio. Today, we'll take a slight detour from our regular material to explore the Python programming language. While this lecture is optional, it's highly recommended as it will provide you with valuable insights into Python, especially in the context of extending GNU Radio's functionalities.

(Refer Slide Time: 01:52)

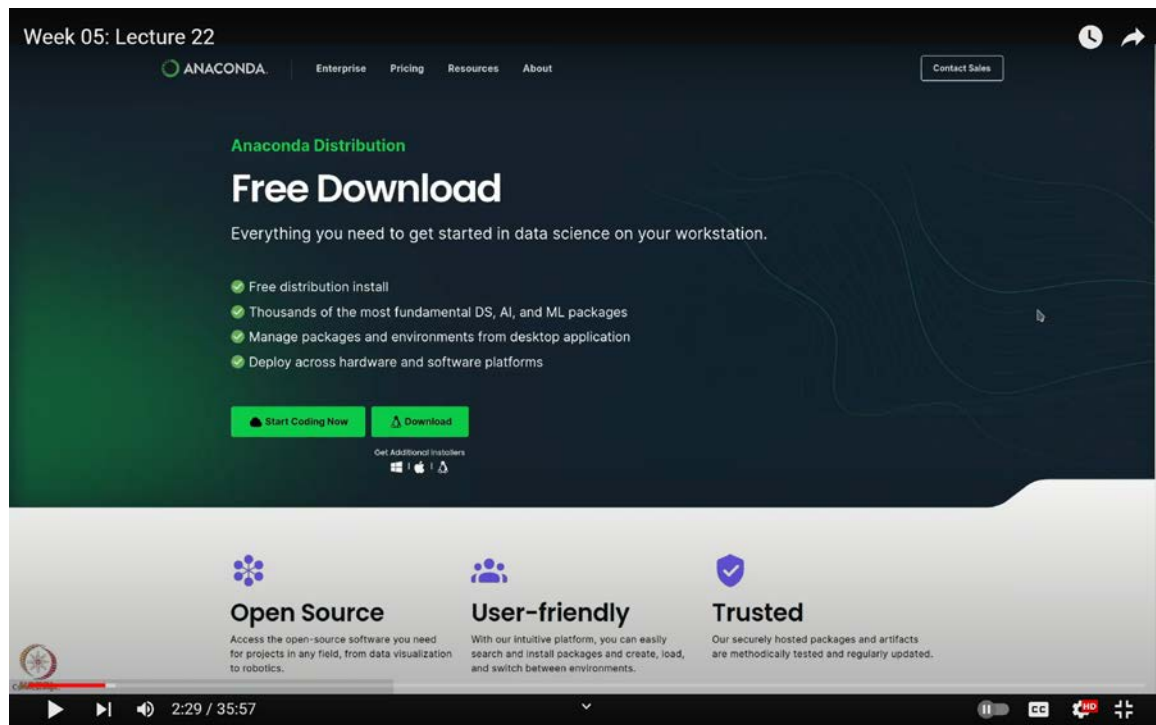


As we've mentioned before, GNU Radio extensively utilizes Python. By learning Python, you can enhance GNU Radio's capabilities or implement functionalities that might not be easily achievable with the built-in blocks. For instance, certain processing tasks can be

more efficiently executed through custom code rather than relying solely on predefined blocks. In this session, we'll offer a brief introduction to Python, focusing specifically on writing extensions for GNU Radio. While this is not a mandatory part of the course, I strongly encourage you to follow along. Additionally, I'll provide some references at the end that will help you expand your Python knowledge.

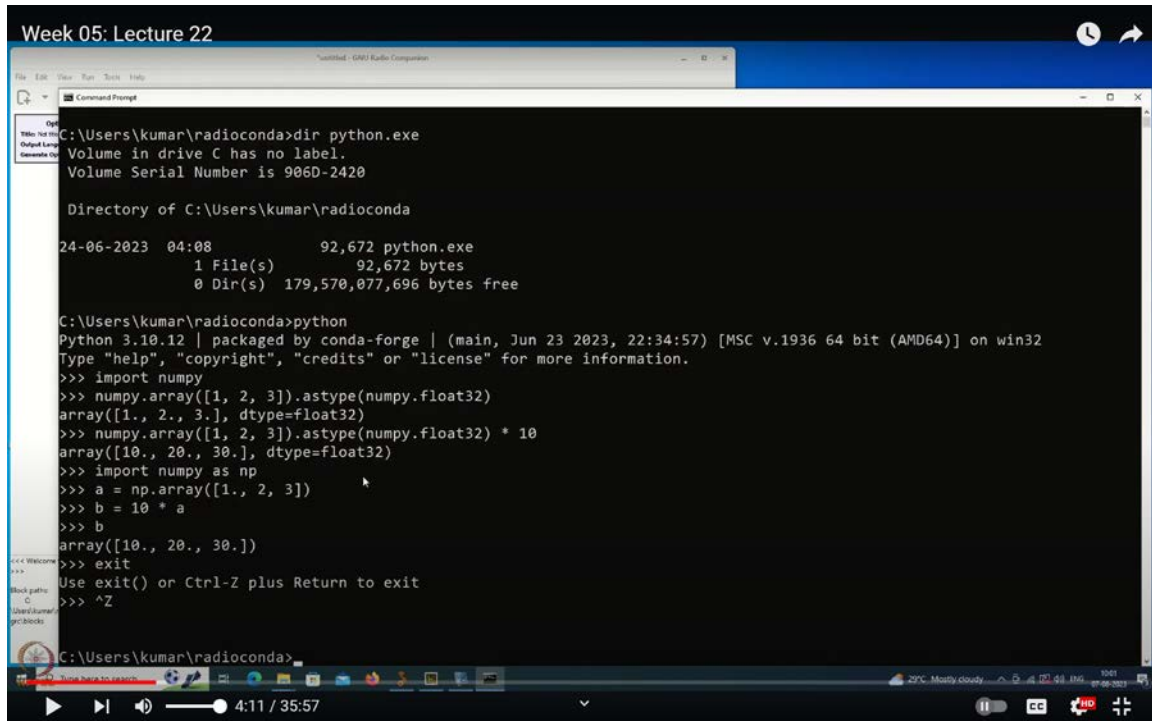
If you're interested in following along with today's lecture but don't have Python installed or prefer not to install it, you can use Google Colaboratory. Simply search for Google Colab, and you'll have access to a Jupyter notebook environment where you can write and execute Python code. This platform supports most features necessary for a Python-based scientific computing experience.

(Refer Slide Time: 02:29)



For those who prefer a full-fledged Python installation, I recommend downloading the Anaconda distribution. Anaconda comes packed with numerous packages that are extremely useful for Python development. In fact, the RadioConda installer for GNU Radio, which some of you may have used, is based on the Anaconda Python distribution.

(Refer Slide Time: 04:11)



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The user is in the directory `C:\Users\kumar\radioconda`. The command `dir python.exe` is executed, showing the file `python.exe` with a size of 92,672 bytes. The command `python` is then executed, starting the Python 3.10.12 interpreter. The user enters `import numpy`, `numpy.array([1, 2, 3]).astype(numpy.float32)`, `numpy.array([1, 2, 3]).astype(numpy.float32) * 10`, `import numpy as np`, `a = np.array([1, 2, 3])`, `b = 10 * a`, and `b`. The output shows the array `array([10., 20., 30.])`. The user then enters `exit` and `^Z` to exit the prompt.

```
Week 05: Lecture 22

C:\Users\kumar\radioconda>dir python.exe
Volume in drive C has no label.
Volume Serial Number is 906D-2420

Directory of C:\Users\kumar\radioconda

24-06-2023  04:08                92,672 python.exe
               1 File(s)                92,672 bytes
               0 Dir(s) 179,570,077,696 bytes free

C:\Users\kumar\radioconda>python
Python 3.10.12 | packaged by conda-forge | (main, Jun 23 2023, 22:34:57) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.array([1, 2, 3]).astype(numpy.float32)
array([1., 2., 3.], dtype=float32)
>>> numpy.array([1, 2, 3]).astype(numpy.float32) * 10
array([10., 20., 30.], dtype=float32)
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> b = 10 * a
>>> b
array([10., 20., 30.])
>>> exit
Use exit() or Ctrl-Z plus Return to exit
>>> ^Z

C:\Users\kumar\radioconda>
```

If you've installed GNU Radio on Windows using RadioConda, you'll already have a Python environment set up. To access it, open the command prompt by typing "cmd" in the Windows search bar, and navigate to the directory where RadioConda is installed. Once there, you'll find a file named "python.exe." By typing `python` and pressing Enter, you'll enter the Python prompt, where you can start coding along with today's lecture. This setup includes essential libraries like NumPy, which we will use today.

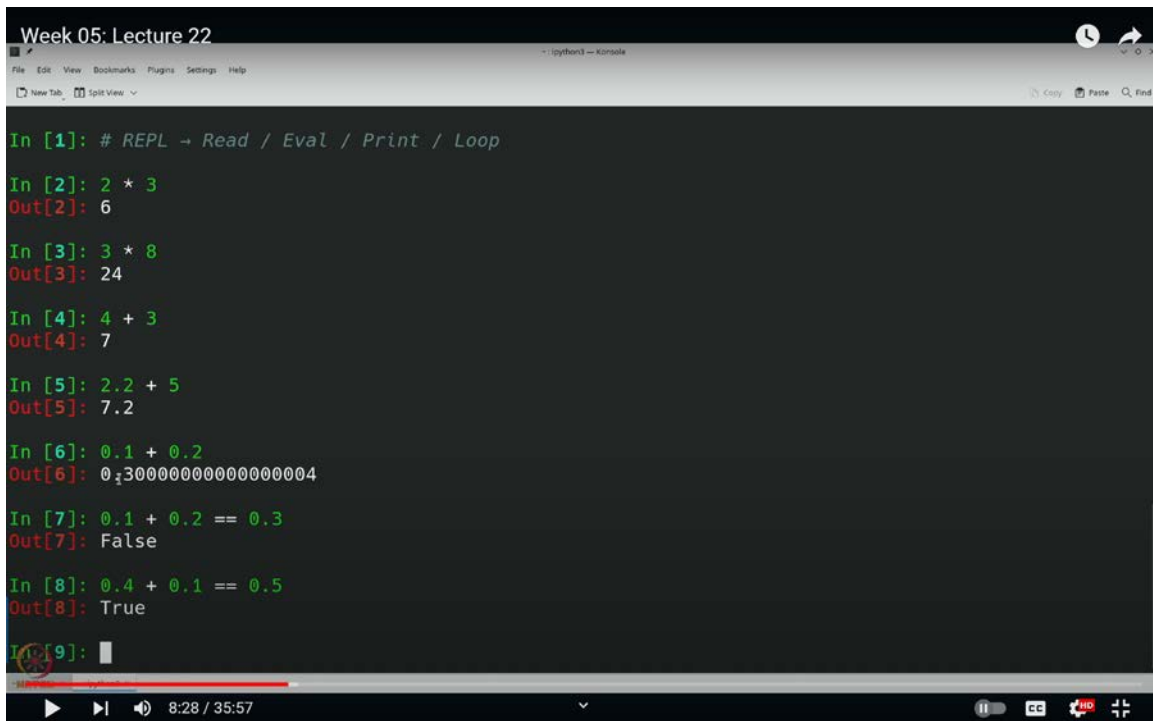
To illustrate, let's create a NumPy array as a float and perform some basic operations. You'll see that all the NumPy features are readily available. One minor limitation is that you might not have autocomplete in this environment, so I suggest importing NumPy as `np` to simplify your coding. This way, you won't need to type out "numpy" each time. Once you're done, you can exit the Python prompt by typing `Ctrl + Z` followed by `Enter`.

For those using macOS or Linux distributions like Ubuntu or Debian, Python is usually pre-installed. You can access it by typing `python3` in your terminal. This command will

launch Python 3 and provide access to all necessary features. For a more user-friendly experience, especially with features like autocomplete, I recommend using ``ipython3`` if it's installed. This enhanced interface allows you to autocomplete commands and easily browse your command history, making your coding experience more efficient. However, if you prefer the regular Python prompt, that's perfectly fine. In this lecture, I'll be using the IPython prompt, but rest assured, the code will be identical whether you use Python or IPython. Let's dive into the code and explore the power of Python in the context of GNU Radio!

The only difference between using ``ipython3`` and regular Python is that ``ipython3`` offers enhanced syntax highlighting and autocomplete. Now, let's explore some features of the Python programming language that will be particularly useful when working with GNU Radio.

(Refer Slide Time: 08:28)

A screenshot of a video player interface. The video title is "Week 05: Lecture 22". The video player shows a terminal window with the following text:

```
In [1]: # REPL -> Read / Eval / Print / Loop
In [2]: 2 * 3
Out[2]: 6
In [3]: 3 * 8
Out[3]: 24
In [4]: 4 + 3
Out[4]: 7
In [5]: 2.2 + 5
Out[5]: 7.2
In [6]: 0.1 + 0.2
Out[6]: 0.30000000000000004
In [7]: 0.1 + 0.2 == 0.3
Out[7]: False
In [8]: 0.4 + 0.1 == 0.5
Out[8]: True
In [9]:
```

The video player controls at the bottom show a play button, a progress bar at 8:28 / 35:57, and other standard video controls.

Before we dive in, it's important to understand that the interface we're using to type commands is known as a REPL, which stands for "Read, Eval (or Evaluate), Print, Loop."

The REPL reads a command, evaluates it, prints the result on the screen, and then loops back to read the next command. REPLs are incredibly convenient for quickly testing small snippets of code to ensure they work before integrating them into a larger program.

In this lecture, we'll focus on REPL-based evaluations in Python to demonstrate some of the language's basic features. Let's start with comments in Python, which are indicated by a hash symbol (`#`). Anything following a hash on a line is ignored by the interpreter, making it a useful tool for adding explanations or notes within your code.

Now, let's use Python as a simple calculator. For instance, typing `2 * 3` returns `6`, and `3 * 8` returns `24`. These are basic multiplication operations. Similarly, `4 + 3` yields `7`. All of these are integer operations. But what happens if we introduce a decimal? For example, `2.2 + 5` results in a decimal value.

It's important to note that decimals in Python (and consequently in GNU Radio, since it's built on Python) use the `float` data type. This can lead to some unexpected results, such as when adding `0.1 + 0.2`. Instead of returning exactly `0.3`, Python might output `0.30000000000000004`. This is a well-known issue with floating-point arithmetic, where precision is sometimes lost due to the way numbers are represented in memory.

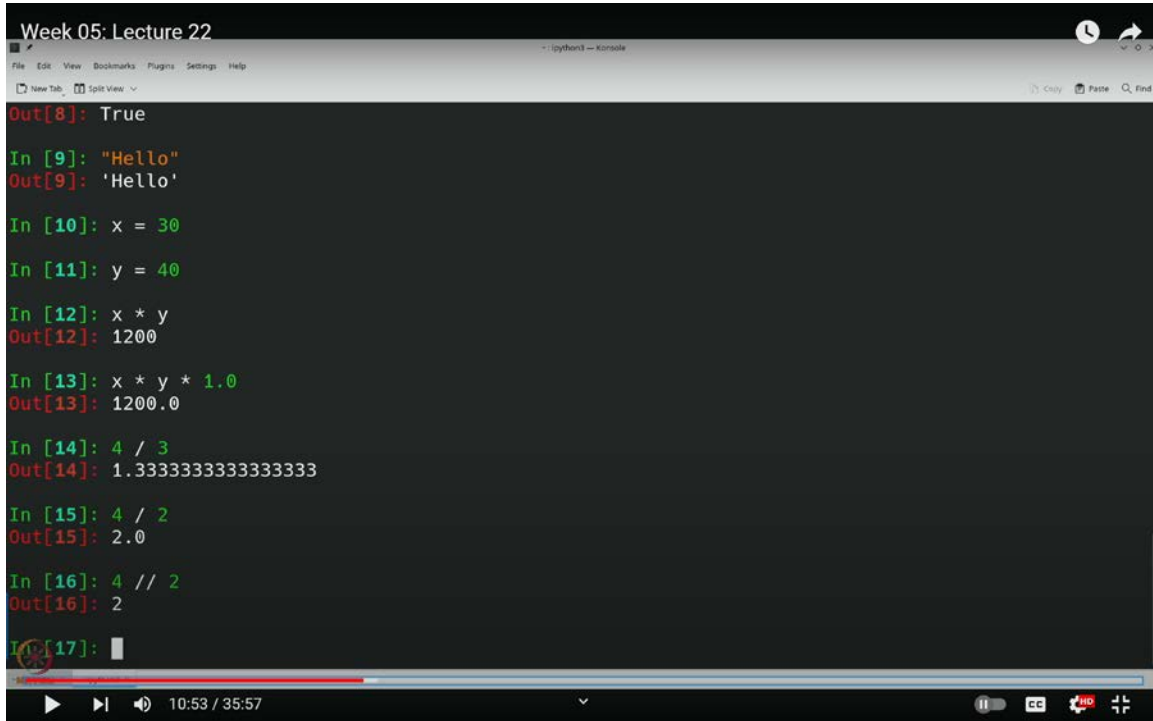
If you try to compare the result of `0.1 + 0.2` with `0.3` using the `==` operator, Python will return `False`. This occurs because floating-point addition often fails to capture the exact precision, leading to minor discrepancies. While this behavior might not always cause problems, such as when `0.4 + 0.1 == 0.5` returns `True`, it's something you should be cautious about when working with floating points.

So far, we've discussed two basic data types: integers and floats. Now, let's introduce another fundamental data type: strings. For example, if you type `"hello"`, Python outputs `'hello'`. In Python, strings can be enclosed in either single or double quotes.

In the `ipython` interface, you'll notice that when you input an expression, the result is printed with an "Out" label. This feature is specific to `ipython`; in regular Python, the evaluated value is simply printed below the expression without any additional labels.

Next, let's talk about variables. If you assign `x = 30` and `y = 40`, then multiply them by typing `x * y`, Python calculates the product and outputs `1200`. This is an assignment operation, where `x = 30` stores the value `30` in the variable `x`, and `y = 40` stores `40` in `y`. The expression `x * y` is then evaluated as `30 * 40`, resulting in `1200`.

(Refer Slide Time: 10:53)

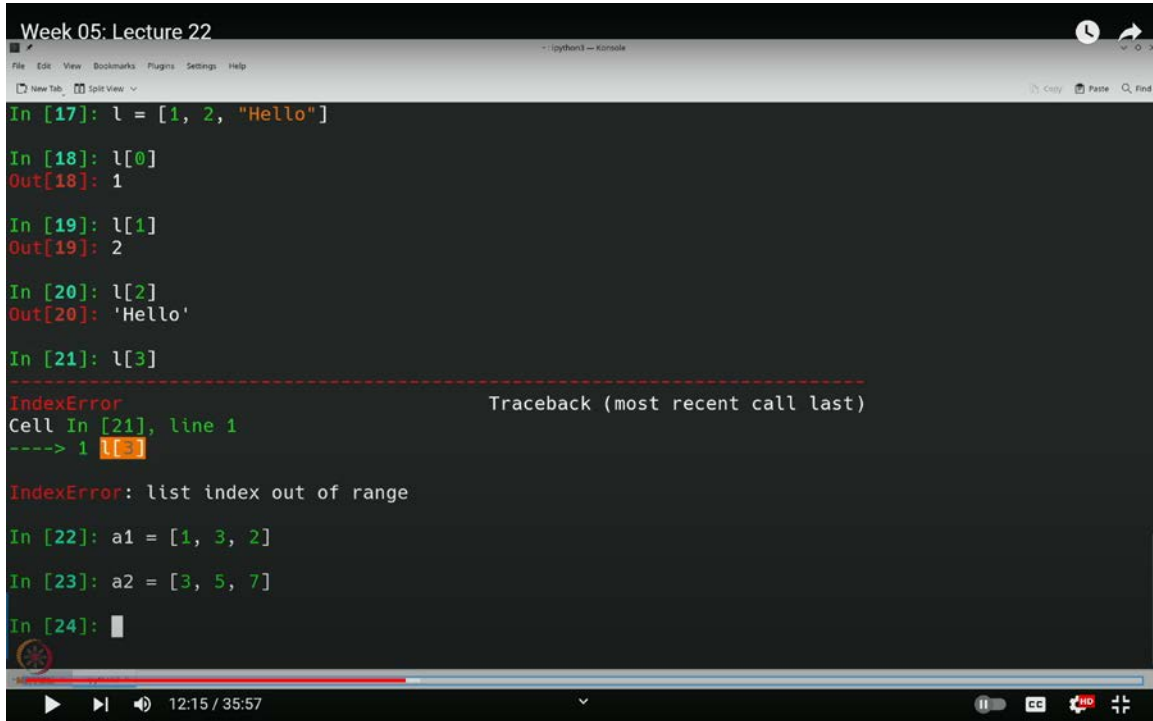
A screenshot of a Jupyter Notebook interface. The title bar at the top says "Week 05: Lecture 22". The interface shows a series of input-output pairs. The first output is "Out[8]: True". The next input is "In [9]: 'Hello'" and the output is "Out[9]: 'Hello'". Then, "In [10]: x = 30" and "In [11]: y = 40" are shown. This is followed by "In [12]: x * y" with output "Out[12]: 1200". Then "In [13]: x * y * 1.0" with output "Out[13]: 1200.0". Next is "In [14]: 4 / 3" with output "Out[14]: 1.3333333333333333". Then "In [15]: 4 / 2" with output "Out[15]: 2.0". Finally, "In [16]: 4 // 2" with output "Out[16]: 2". The last line shows "In [17]:" with a cursor. At the bottom, a video player interface is visible, showing a progress bar at 10:53 / 35:57 and various control icons.

This operation is an example of integer multiplication. However, if you were to multiply by a floating-point number, such as `x * 1.0`, Python automatically converts the result to a float. This behavior is particularly relevant when performing division. For instance, `4 / 3` returns `1.3333`, which is a floating-point result, as expected for the division of two integers.

Interestingly, even `4 / 2` returns `2.0`, indicating that Python treats division results as floating-point numbers by default. If you specifically want integer division, you can use the double slash operator (`//`). For example, `4 // 2` returns `2`, which is an integer result. This distinction is crucial, especially in contexts where integer division is required, such as in certain signal processing tasks.

We've just explored some very basic data types and operations in Python. If you wish to dive deeper into functions, modules, or other advanced topics, there are numerous resources available. However, for our purposes, particularly in the context of working with GNU Radio, we are primarily interested in these fundamental concepts.

(Refer Slide Time: 12:15)



```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
In [17]: l = [1, 2, "Hello"]
In [18]: l[0]
Out[18]: 1
In [19]: l[1]
Out[19]: 2
In [20]: l[2]
Out[20]: 'Hello'
In [21]: l[3]
-----
IndexError                                Traceback (most recent call last)
Cell In [21], line 1
----> 1 l[3]

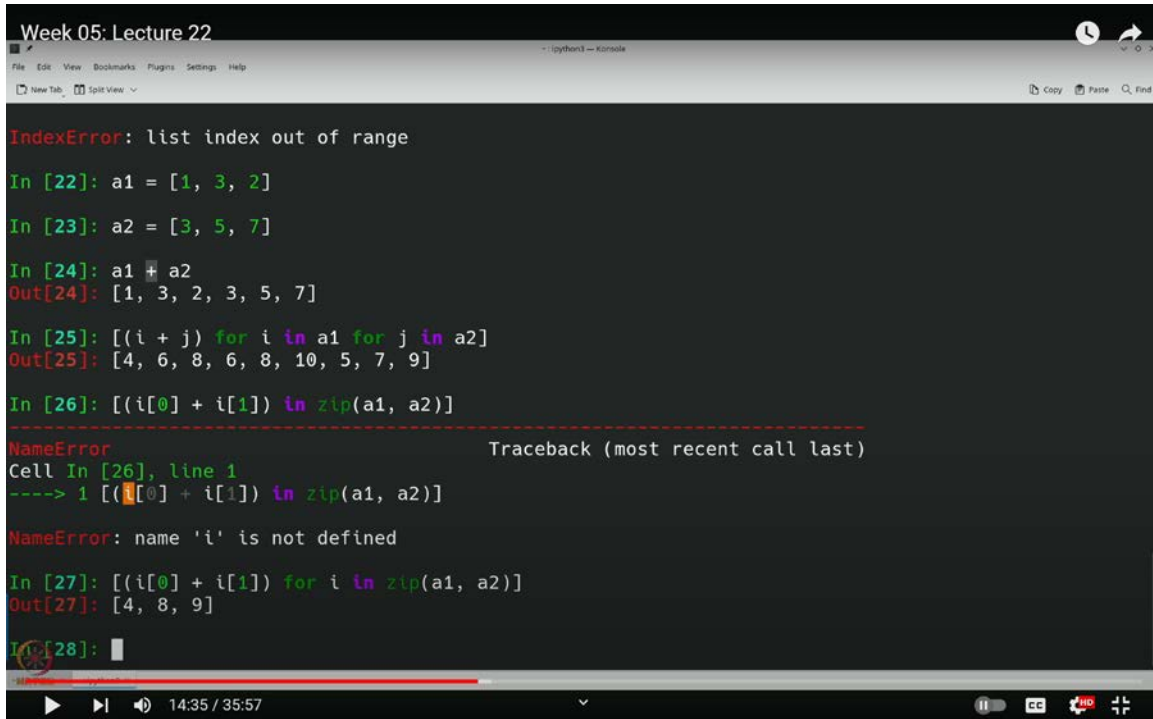
IndexError: list index out of range
In [22]: a1 = [1, 3, 2]
In [23]: a2 = [3, 5, 7]
In [24]:
```

The next data type we will discuss is the list. Consider the following example: `l = [1, 2, 'hello']`. This command creates a Python list containing the elements `1`, `2`, and `'hello'`. Lists in Python are versatile structures, allowing you to store a collection of items of different types. For instance, if you access the list using `l[0]`, Python returns `1`; `l[1]` returns `2`; and `l[2]` returns `'hello'`. However, attempting to access `l[3]` will result in an "IndexError: list index out of range" because the list only has three elements indexed from 0 to 2.

You can think of lists as dynamic arrays, but with a key distinction: in Python, lists can hold elements of any data type, including other lists. This flexibility is one of the reasons why lists are so powerful in Python. Additionally, lists come with many useful features,

you can easily append elements, remove elements, insert elements at specific positions, and more. While there are other data structures like sets and dictionaries in Python, we will skip over them for now to stay focused on our current discussion.

(Refer Slide Time: 14:35)



```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
Copy Paste Find

IndexError: list index out of range

In [22]: a1 = [1, 3, 2]
In [23]: a2 = [3, 5, 7]
In [24]: a1 + a2
Out[24]: [1, 3, 2, 3, 5, 7]
In [25]: [(i + j) for i in a1 for j in a2]
Out[25]: [4, 6, 8, 6, 8, 10, 5, 7, 9]
In [26]: [(i[0] + i[1]) in zip(a1, a2)]

-----
NameError                                Traceback (most recent call last)
Cell In [26], line 1
----> 1 [(i[0] + i[1]) in zip(a1, a2)]

NameError: name 'i' is not defined

In [27]: [(i[0] + i[1]) for i in zip(a1, a2)]
Out[27]: [4, 8, 9]
In [28]:
```

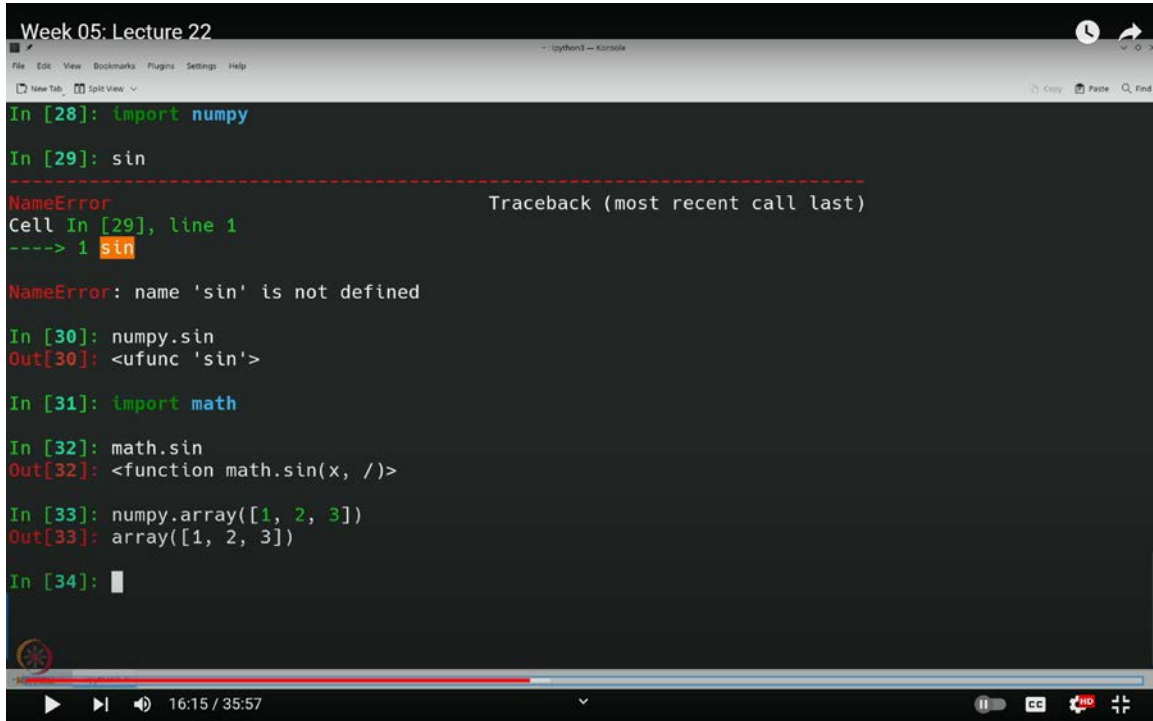
However, when it comes to numerical computations, lists present a significant challenge. Let's consider an example where you have two lists: `a1 = [1, 3, 2]` and `a2 = [3, 5, 7]`. You might expect that adding these lists together would yield an element-wise sum, such as `[1+3, 3+5, 2+7]` resulting in `[4, 8, 9]`. Unfortunately, this is not how Python handles list addition. Instead, `a1 + a2` simply concatenates the two lists, producing `[1, 3, 2, 3, 5, 7]`.

This behavior stems from the fact that Python lists are generic containers, designed to hold multiple objects of different types. Consequently, when using the `+` operator, Python defaults to concatenation rather than element-wise arithmetic. Furthermore, performing arithmetic operations with lists is not recommended because it can be inefficient and slow.

If you truly need to perform element-wise addition, Python provides a workaround using

list comprehensions. For example, you could write `[i + j for i, j in zip(a1, a2)]`, which would correctly produce `[4, 8, 9]`. However, this method can be cumbersome and inefficient for larger datasets, making it impractical for scientific and engineering applications, such as those in GNU Radio.

(Refer Slide Time: 16:15)



```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
In [28]: import numpy
In [29]: sin
-----
NameError                                Traceback (most recent call last)
Cell In [29], line 1
----> 1 sin

NameError: name 'sin' is not defined

In [30]: numpy.sin
Out[30]: <ufunc 'sin'>

In [31]: import math
In [32]: math.sin
Out[32]: <function math.sin(x, /)>

In [33]: numpy.array([1, 2, 3])
Out[33]: array([1, 2, 3])

In [34]:
```

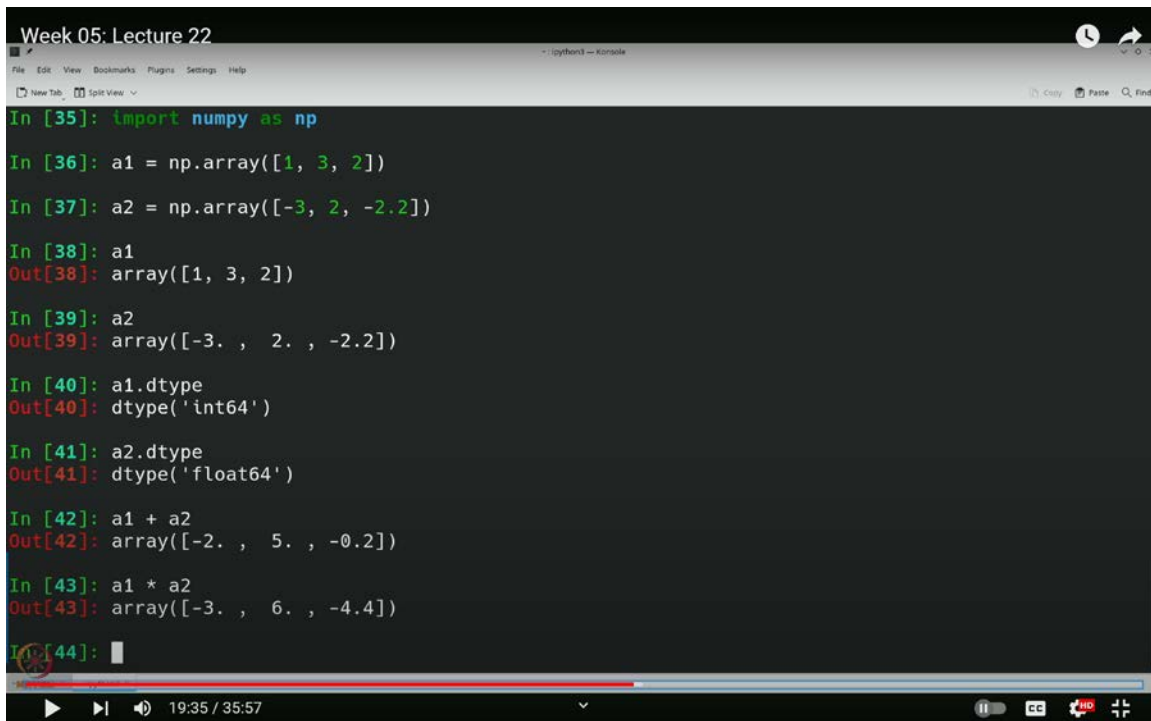
To efficiently handle numerical computations, we use a specialized module called `numpy`. `numpy` is not included in the default Python distribution, but it is bundled with GNU Radio and is an essential tool for numerical computing. Let's take a quick look at how to use `numpy`.

To start, you need to import the `numpy` module by typing `import numpy`. In Python, the `import` command allows you to access features of libraries or modules that are not available by default. Once imported, you can access `numpy`'s functionalities by prefixing them with `numpy.`. For instance, to use the `sin` function from `numpy`, you would type `numpy.sin()`.

Python is designed with careful attention to namespaces, which prevents conflicts between

similarly named functions from different modules. For example, Python has a built-in `math` module that also contains a `sin` function. The key difference is that `numpy.sin()` operates on arrays, while `math.sin()` only works with individual numbers. To avoid confusion, Python requires you to specify the module name when calling a function, ensuring you get the correct version.

(Refer Slide Time: 19:35)



```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
In [35]: import numpy as np
In [36]: a1 = np.array([1, 3, 2])
In [37]: a2 = np.array([-3, 2, -2.2])
In [38]: a1
Out[38]: array([1, 3, 2])
In [39]: a2
Out[39]: array([-3. ,  2. , -2.2])
In [40]: a1.dtype
Out[40]: dtype('int64')
In [41]: a2.dtype
Out[41]: dtype('float64')
In [42]: a1 + a2
Out[42]: array([-2. ,  5. , -0.2])
In [43]: a1 * a2
Out[43]: array([-3. ,  6. , -4.4])
In [44]:
```

However, constantly typing `numpy.` can be tedious, so a common practice is to import `numpy` with an alias, such as `import numpy as np`. This allows you to use `np.` as a shorthand when calling `numpy` functions, making your code more concise and readable.

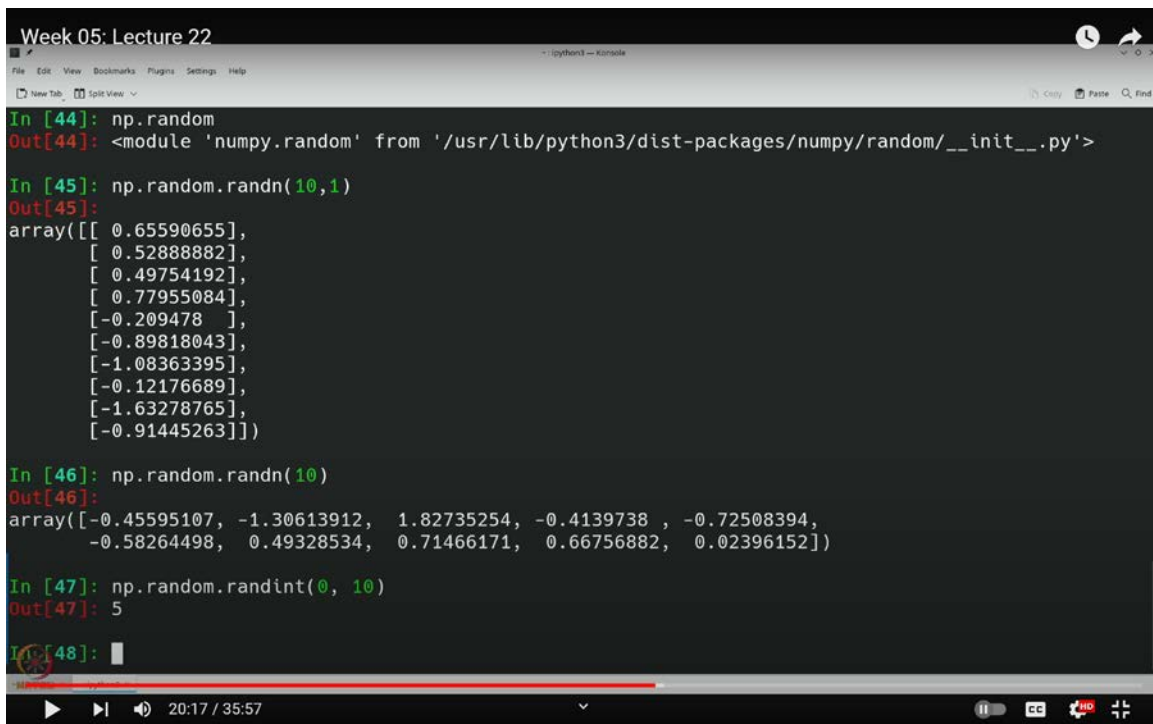
Now, let's create some `numpy` arrays. If you define `a1 = np.array([1, 3, 2])` and `a2 = np.array([-3, 2, -2.2])`, you'll notice that `a1` is a `numpy` array with integers, while `a2` is a `numpy` array with floating-point numbers. A subtle difference from Python lists is that `numpy` arrays are designed to hold elements of the same data type. For example, the data type of `a1` is `int64`, while the data type of `a2` is `float64`. Despite these differences, `numpy` allows you to perform arithmetic operations between arrays,

automatically promoting the integer array to a floating-point array when necessary.

This consistency in data types and the ability to efficiently perform arithmetic operations on large datasets make `numpy` an indispensable tool for scientists and engineers working with GNU Radio. By leveraging `numpy`, you can perform complex numerical computations with ease and accuracy.

Let's start by performing a simple addition operation. When you add `a1` and `a2`, the result is exactly as expected. The element-wise addition proceeds as follows: 1 adds to -3, 3 adds to 2, and 2 adds to -2.2. This yields the results $1 - 3 = -2$, $3 + 2 = 5$, and $2 - 2.2 = -0.2$. This is precisely the outcome we anticipate. Similarly, if you multiply `a1` by `a2`, you get element-wise multiplication: 1 times -3, 3 times 2, and 2 times -2.2. This demonstrates how convenient and efficient numpy arrays are for numerical computations, particularly when used as components within GNU Radio.

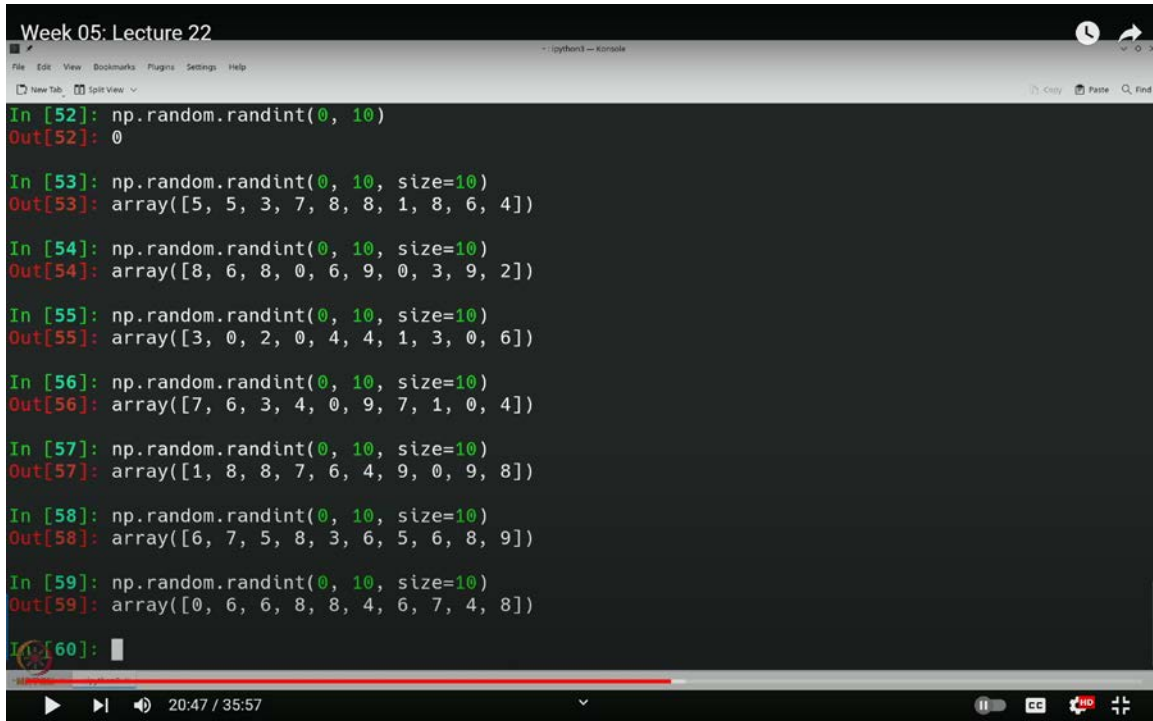
(Refer Slide Time: 20:17)

The image is a screenshot of a video player window. The title bar at the top reads "Week 05: Lecture 22". The video content shows a Jupyter Notebook interface with a dark background. The notebook has three input cells and their corresponding outputs. The first cell contains `np.random` and the output is `<module 'numpy.random' from '/usr/lib/python3/dist-packages/numpy/random/__init__.py'>`. The second cell contains `np.random.randn(10,1)` and the output is a 10x1 array of random floats. The third cell contains `np.random.randn(10)` and the output is a 1D array of 10 random floats. The fourth cell contains `np.random.randint(0, 10)` and the output is the integer 5. The video player controls at the bottom show a progress bar at 20:17 / 35:57.

Numpy also provides several convenient functions that we should explore. For instance, if you want to generate random numbers, you can use `np.random`, a sub-module within

numpy. A sub-module is simply a module within another module. To generate 10 Gaussian random variables, you can use `np.random.randn(10, 1)`.

(Refer Slide Time: 20:47)



```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
In [52]: np.random.randint(0, 10)
Out[52]: 0

In [53]: np.random.randint(0, 10, size=10)
Out[53]: array([5, 5, 3, 7, 8, 8, 1, 8, 6, 4])

In [54]: np.random.randint(0, 10, size=10)
Out[54]: array([8, 6, 8, 0, 6, 9, 0, 3, 9, 2])

In [55]: np.random.randint(0, 10, size=10)
Out[55]: array([3, 0, 2, 0, 4, 4, 1, 3, 0, 6])

In [56]: np.random.randint(0, 10, size=10)
Out[56]: array([7, 6, 3, 4, 0, 9, 7, 1, 0, 4])

In [57]: np.random.randint(0, 10, size=10)
Out[57]: array([1, 8, 8, 7, 6, 4, 9, 0, 9, 8])

In [58]: np.random.randint(0, 10, size=10)
Out[58]: array([6, 7, 5, 8, 3, 6, 5, 6, 8, 9])

In [59]: np.random.randint(0, 10, size=10)
Out[59]: array([0, 6, 6, 8, 8, 4, 6, 7, 4, 8])

In [60]:
```

This command generates a two-dimensional array with 10 random Gaussian values. If you only need a single array, you can use `np.random.randn(10)`, which will yield a one-dimensional array. Alternatively, if you want to generate random integers, you can use `np.random.randint(0, 10, size=10)`. This command produces 10 random integers between 0 and 9 (note that 10 is not included).

But what if you want to define a specific sequence or waveform? In numpy, you can achieve this using two approaches: `linspace` and `arange`. Let's say you want to generate a sine wave. You can use `np.linspace` to create a set of numbers. For example, if you set `T = np.linspace(0, 10, 10)`, `T` will contain 10 equally spaced numbers between 0 and 10, inclusive. These numbers would be 0, 1/9, 2/9, 3/9, ..., up to 9/9, effectively dividing the interval into 10 equal parts.

(Refer Slide Time: 22:03)

```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View Copy Paste Find

In [60]: t = np.linspace(0, 10, 10)

In [61]: t
Out[61]: array([ 0.,          1.11111111,  2.22222222,  3.33333333,  4.44444444,
 5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.])

In [62]: t = np.linspace(0, 9, 10)

In [63]: t
Out[63]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

In [64]: t = np.linspace(0, 10, 1000)

In [65]:
```

(Refer Slide Time: 22:38)

```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View Copy Paste Find

9.35935936, 9.36936937, 9.37937938, 9.38938939, 9.3993994 ,
9.40940941, 9.41941942, 9.42942943, 9.43943944, 9.44944945 ,
9.45945946, 9.46946947, 9.47947948, 9.48948949, 9.4994995 ,
9.50950951, 9.51951952, 9.52952953, 9.53953954, 9.54954955,
9.55955956, 9.56956957, 9.57957958, 9.58958959, 9.5995996 ,
9.60960961, 9.61961962, 9.62962963, 9.63963964, 9.64964965,
9.65965966, 9.66966967, 9.67967968, 9.68968969, 9.6996997 ,
9.70970971, 9.71971972, 9.72972973, 9.73973974, 9.74974975,
9.75975976, 9.76976977, 9.77977978, 9.78978979, 9.7997998 ,
9.80980981, 9.81981982, 9.82982983, 9.83983984, 9.84984985,
9.85985986, 9.86986987, 9.87987988, 9.88988989, 9.8998999 ,
9.90990991, 9.91991992, 9.92992993, 9.93993994, 9.94994995,
9.95995996, 9.96996997, 9.97997998, 9.98998999, 10.])

In [66]: t = np.arange(10)

In [67]: t
Out[67]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

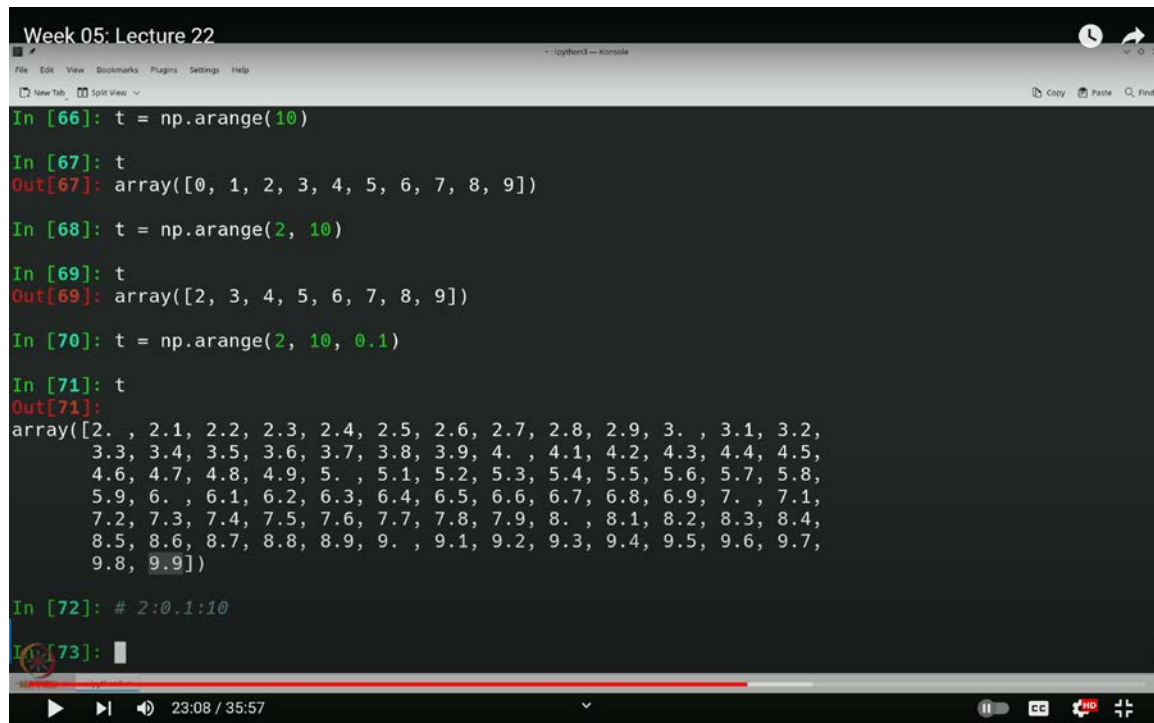
In [68]: t = np.arange(2, 10)

In [69]: t
Out[69]: array([2, 3, 4, 5, 6, 7, 8, 9])

In [70]: t = np.arange(2, 10, 0.1)
```

If you prefer integer spacing, you can adjust the range accordingly. For instance, if you set `T = np.linspace(0, 9, 10)`, the result will be integers spaced evenly between 0 and 9. However, in practice, it's more common to use a higher number of divisions, as finer granularity is often desired for numerical analysis.

(Refer Slide Time: 23:08)

A screenshot of a Jupyter Notebook interface titled "Week 05: Lecture 22". The notebook shows a series of code cells in a dark theme. The first cell (In [66]) defines `t = np.arange(10)`, and the output (Out [67]) is `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`. The second cell (In [68]) defines `t = np.arange(2, 10)`, and the output (Out [69]) is `array([2, 3, 4, 5, 6, 7, 8, 9])`. The third cell (In [70]) defines `t = np.arange(2, 10, 0.1)`, and the output (Out [71]) is a large array of values from 2.0 to 9.9 in increments of 0.1. The fourth cell (In [72]) shows the string representation `# 2:0.1:10`. The fifth cell (In [73]) is empty. At the bottom, a video player interface shows a progress bar at 23:08 / 35:57.

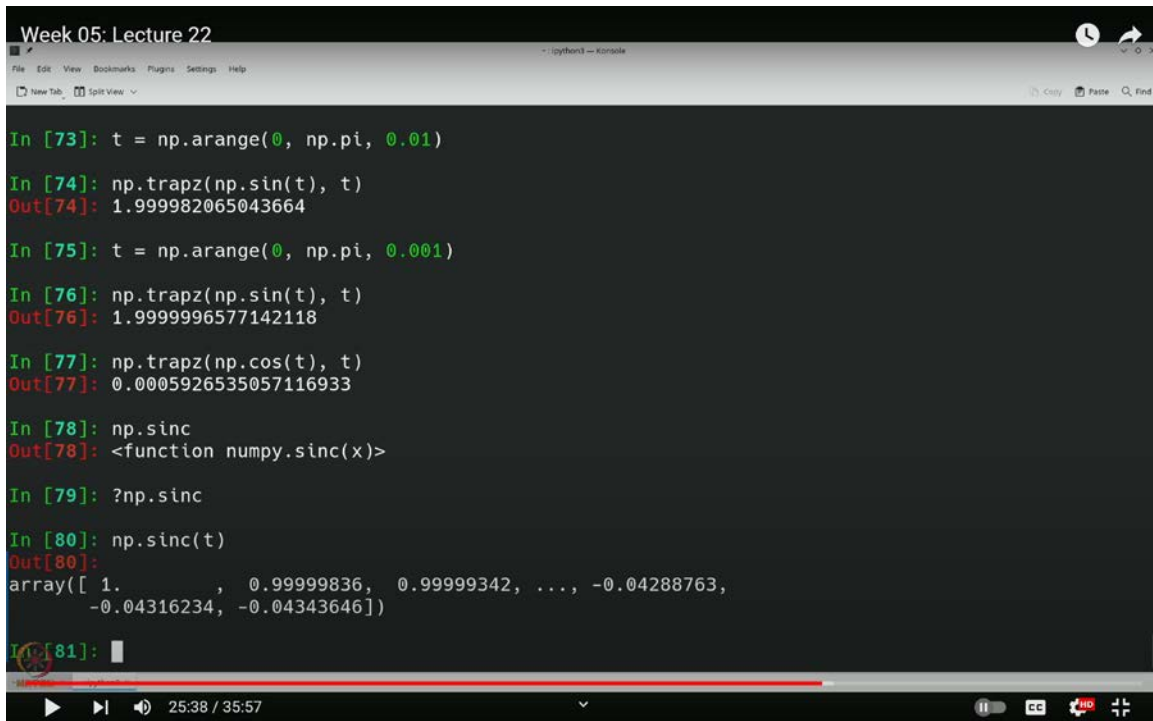
Alternatively, you can use the `arange` function, which offers additional flexibility. For example, `T = np.arange(10)` generates numbers from 0 through 9. If you specify a starting point and an endpoint, such as `T = np.arange(2, 10)`, you get numbers from 2 through 9. Always remember that the last number is not included in the range. You can further refine this by adding a step size. For example, `T = np.arange(2, 10, 0.1)` gives you numbers starting from 2 up to (but not including) 10, with each successive number increasing by 0.1. This approach is similar to MATLAB or Octave's colon operator (e.g., `2:0.1:10`), with the distinction that `np.arange` stops just before reaching the final value.

Now, let's do something practical with these tools by generating a simple waveform. We'll define `T` using `np.arange` from 0 to π with a step size of 0.01: `T = np.arange(0, np.pi,`

`0.01`). This creates an array of values from 0 to π with increments of 0.01.

Next, let's perform a trapezoidal integration of the sine function over this range. We can use `np.trapz` to achieve this. Specifically, `np.trapz(np.sin(T), T)` will numerically integrate `sin(T)` from 0 to π .

(Refer Slide Time: 25:38)



```
Week 05: Lecture 22
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
Copy Paste Find

In [73]: t = np.arange(0, np.pi, 0.01)

In [74]: np.trapz(np.sin(t), t)
Out[74]: 1.999982065043664

In [75]: t = np.arange(0, np.pi, 0.001)

In [76]: np.trapz(np.sin(t), t)
Out[76]: 1.9999996577142118

In [77]: np.trapz(np.cos(t), t)
Out[77]: 0.0005926535057116933

In [78]: np.sinc
Out[78]: <function numpy.sinc(x)>

In [79]: ?np.sinc

In [80]: np.sinc(t)
Out[80]:
array([ 1.          ,  0.99999836,  0.99999342, ..., -0.04288763,
        -0.04316234, -0.04343646])

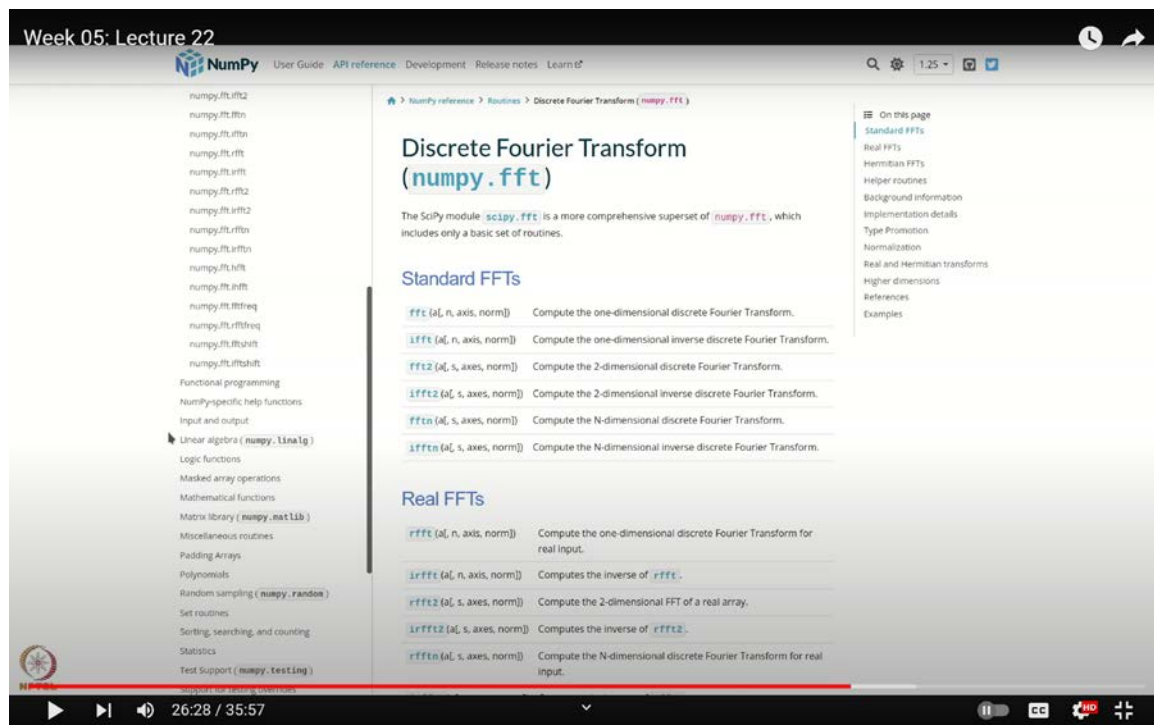
In [81]:
```

Before executing this, it's worth recalling that the exact integral of `sin(T)` from 0 to π is 2. When you run the code, you'll notice that the result is approximately 1.999982, which is very close to 2. If you desire greater accuracy, you can reduce the step size, although this will increase computation time. However, doing so will yield a result even closer to the theoretical value of 2.

Similarly, if you wish to integrate the cosine function, you should recall that $\cos(x)$ oscillates from its maximum at 0 to its minimum at π , and the integral of $\cos(x)$ from 0 to π should indeed sum to zero. When you perform this integration, you get a result very close to 0.0005, which is consistent with the expected outcome.

Numpy offers several convenient functions to simplify your work. One particularly useful function is `np.sinc`. If you type a question mark after `np.sinc`, it will provide you with quick documentation. The `sinc` function computes $\frac{\sin(\pi x)}{\pi x}$. For example, using `np.sinc(T)` will generate a sequence of values representing the sinc function, starting from 0. I am deliberately not covering plotting in this explanation; plotting can be done with the `matplotlib` library, which you can explore further on your own. For now, we can integrate these functionalities within GNU Radio and utilize plotting and other features as needed.

(Refer Slide Time: 26:28)



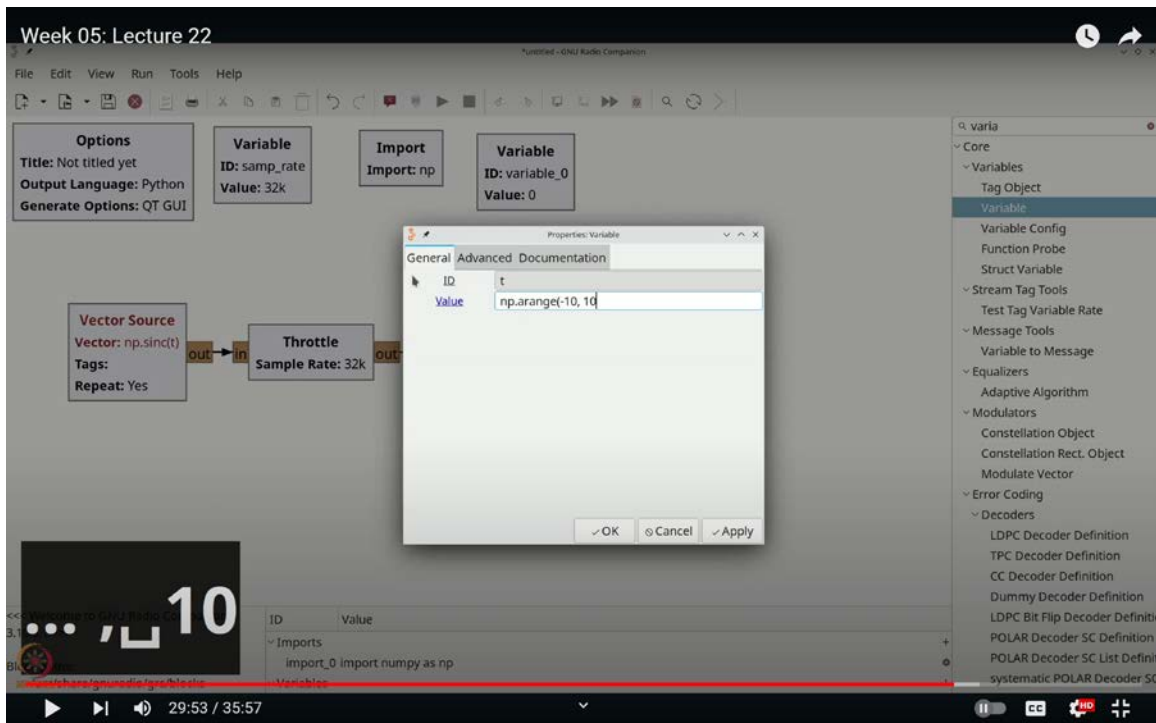
Lastly, if you want to deepen your understanding of numpy, the numpy website is a valuable resource. By visiting the numpy website and navigating to the documentation section, you'll find extensive user guides and an API reference. This documentation provides detailed information about all the features we've discussed, including advanced topics like the discrete Fourier transform (FFT) and other functionalities. While proficiency in numpy is not mandatory, it significantly enhances your ability to use GNU Radio effectively, especially in scenarios where Python's capabilities are advantageous. For a

broad understanding of Python, you might also consider exploring general Python tutorials.

The Python tutorial is accessible on the Python website. Simply navigate to the documentation section and select the tutorial. Carefully working through this tutorial will be extremely beneficial and educational. Although it's not mandatory, it will certainly make your life much easier if you need to implement certain functionalities of GNU Radio using Python.

Now, let's explore how to integrate Python within GNU Radio. There are several approaches to achieve this. One method is to use the available Python blocks. To find these, press **Ctrl+F** or **Command+F** and search for "Python." You will come across three options: Python block, Python module, and Python snippet. For our purposes, we will focus on the Python block and Python module. The Python snippet, which allows modification of existing blocks, is not part of our current discussion.

(Refer Slide Time: 29:53)



Before diving into Python blocks and modules, let's revisit the **import** function. Again,

press `Ctrl+F` or `Command+F` and type "import." By double-clicking on the import option, you can utilize it to bring in libraries. Instead of just `import numpy`, we can use `import numpy as np` for convenience. This allows us to leverage numpy's functionalities more efficiently.

Let's now visualize this by creating a vector source. Press `Ctrl+F` or `Command+F` and type "vector" to find and select the vector source. Next, find and add a throttle by pressing `Ctrl+F` or `Command+F` and typing "throttle." Set the throttle to float for ease of use. Similarly, add a time sync by searching for "time sync," and also set this to float. Finally, add a grid and enable auto-scaling. Connect all these components together to complete the setup.

Let's proceed by creating a vector source that generates a sync signal. Double-click on the vector source block and enter `np.sync` as the function. For the input, specify a variable named `t`. If you click "OK," you may encounter an error. To identify the issue, double-click on the error message, which will indicate that `t` is not defined.

(Refer Slide Time: 30:43)

The screenshot displays a video player interface for a software setup. The main window shows a block diagram with the following components and settings:

- Options:** Title: Not titled yet, Output Language: Python, Generate Options: QT GUI.
- Variable:** ID: samp_rate, Value: 32k.
- Import:** Import: np.
- Variable:** ID: t, Value: np.arange(-10, 10, ...).

The block diagram shows a **Vector Source** block with the following settings:

- Vector: np.sync(t)
- Tags: (empty)
- Repeat: Yes

The **Vector Source** is connected to a **Throttle** block with the following settings:

- Sample Rate: 32k

A **Properties: QT GUI Time Sink** dialog is open, showing the following settings:

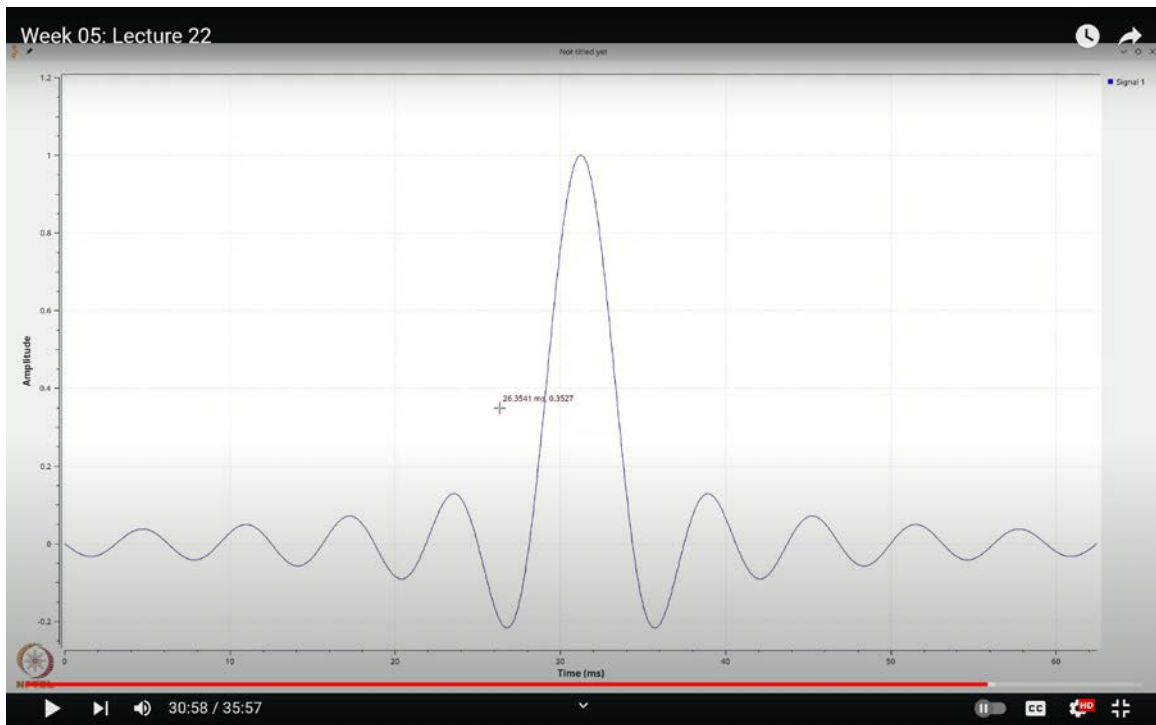
- General: Type (Float), Name (t), Y Axis Label (Amplitude), Y Axis Unit (sin), Number of Points (len(t)), Sample Rate (samp_rate), Grid (Yes), Autoscale (Yes), Y_min (-1), Y_max (1), Number of Inputs (1).
- Advanced: (empty)
- Trigger: (empty)
- Config: (empty)
- Documentation: (empty)

A search bar on the right shows the text "len(t)" and a list of variables including "len(t)".

Remember, GNU Radio Companion generates Python code where ``np.sync`` is used with ``t``, but ``t`` has not been defined yet. Instead of manually editing the Python code, it's more convenient to handle this directly within GNU Radio Companion by using a variable block.

Press ``Ctrl+F`` or ``Command+F`` and type "variable" to find and insert a variable block. Double-click on this block to configure it. Name the variable ``t`` and set its value using ``np.arange(-10, 10, 0.01)``. This will generate a range of values from -10 to 10 with a step size of 0.01.

(Refer Slide Time: 30:58)



Now, execute the flow graph. Save and run it, and you will see the sync signal varying. The variation occurs because the width of the time sync and the sync signal are not the same. To correct this, ensure that the time sync block is configured to display the sync signal fully. Double-click on the time sync block and adjust its settings.

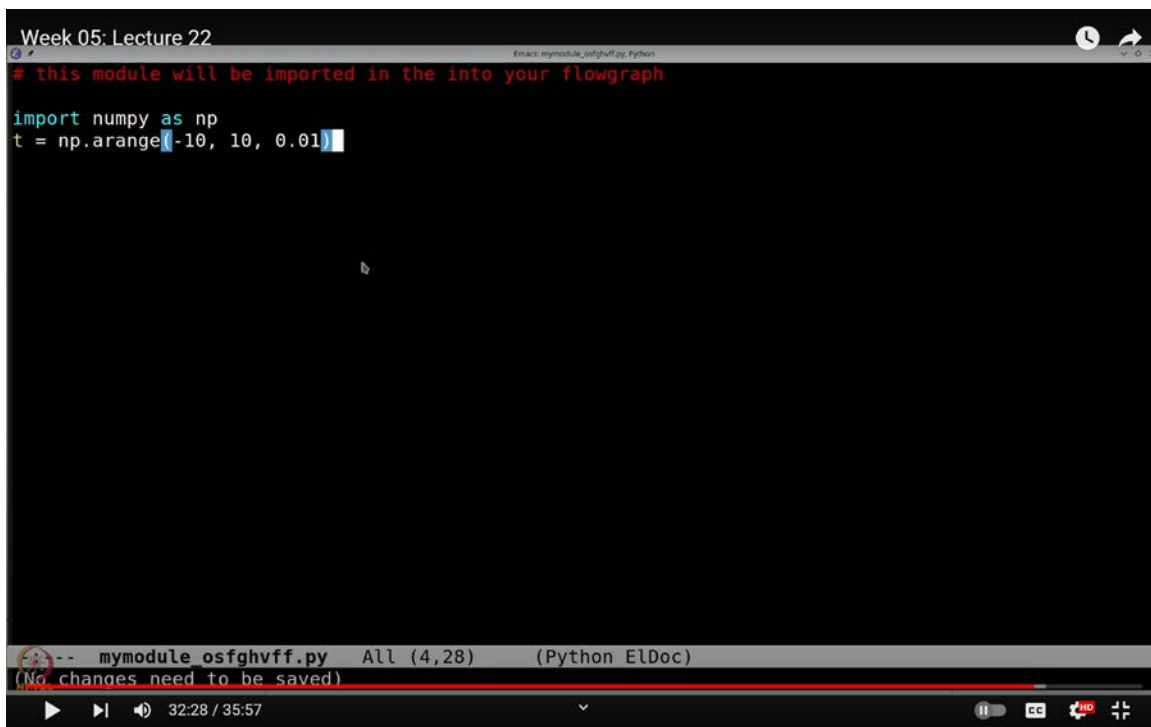
Python has a function called ``len`` that returns the length of a variable. Since the length of the sync signal should match the length of ``t``, use ``len(t)`` to set the number of points for the sync signal. This adjustment will ensure that the sync signal aligns correctly with the

time sync, as both will now have the same length.

As you can see, this setup greatly simplifies the process. Now, let's discuss the Python module. If you need to write a custom function to modify a specific aspect of your flow graph, the Python module becomes extremely valuable.

To start, add a Python module block to your flow graph. Double-click on the Python module block to open its properties. You can rename the module to something more convenient, such as "my_module." By selecting "Open in Editor," you can open the module in any preferred code editor like Visual Studio Code or Spyder, which will provide syntax highlighting and writing assistance.

(Refer Slide Time: 32:28)



```
Week 05: Lecture 22
# this module will be imported in the into your flowgraph
import numpy as np
t = np.arange(-10, 10, 0.01)
```

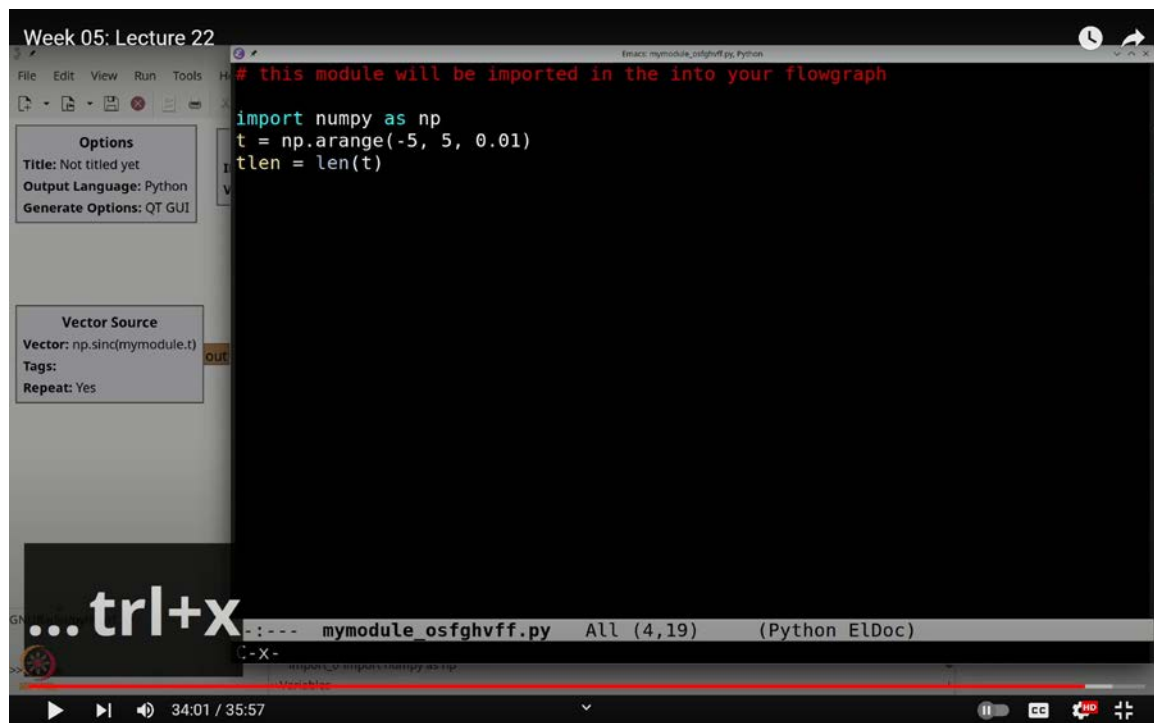
-- mymodule_osfghvff.py All (4,28) (Python ElDoc)
(No changes need to be saved)

32:28 / 35:57

In the editor, you can define your variable `t` as follows: `import numpy as np` and then set `t` with `np.arange(-10, 10, 0.01)`. You can write any additional functions or variables here, and these will be accessible within your flow graph. This is particularly useful for writing multi-line code or performing complex computations without cluttering the flow graph itself.

Once you have made your edits, save and close the editor, then click "OK." Since you've moved the definition of `t` into the module, you can now remove the `t` variable and the `import numpy as np` from the main flow graph. To ensure everything works, re-add the import statement for the sync function if necessary. Replace the previous `import` and `t` references with `my_module.t` where applicable.

(Refer Slide Time: 34:01)



If you find it cumbersome to keep typing `my_module.t`, you can simplify things by modifying your module. For instance, you can create a new variable `tlen`, which represents the length of `t`. Update your code to reflect this change: replace occurrences of `t` with `tlen` where necessary, then save and exit. Now, you can use `my_module.tlen` in place of repeatedly typing `my_module.t`. When you execute your flow graph, it should work as expected.

If you adjust the range of `t` to go from, say, -5 to 5, you will see a shorter sync with fewer cycles. This demonstrates how you can use module features to streamline your code and integrate it effectively into your flow graph.

This overview provides a brief introduction to using Python with GNU Radio. While this is a quick snapshot, Python is a versatile programming language with extensive capabilities that extend beyond the scope of this course. Although learning Python is not mandatory for using GNU Radio, acquiring this knowledge can significantly ease the implementation of certain functionalities and enhance your ability to perform advanced tasks. I strongly encourage you to learn Python thoroughly, as it will not only aid in mastering GNU Radio extensions but also prove beneficial in various other areas. Thank you.