University of Southampton
Institute of Sound and Vibration Research
MSc Sound and Vibration Studies
Tutorial on Python Programming Language

September 26, 2017

Contents

1 Setting Up

This Tutorial session aims to provide a brief revision of programming concepts using Python. We will use the Anaconda Python Distribution, a Python-language-based "ecosystem" which is free, already contains the essential packages for scientific computing and can easily be downloaded from the internet[1]. Anaconda should already be available when you log in University computers, but you are free to use your own personal computer if you wish to. If this is your first time installing Python in your personal computer, we urge you to adopt Python version 3.6 instead of version 2.7.

The Anaconda Python Distribution includes a command-line package manager application called `conda` that allows the user to quickly install, run, and update packages and their dependencies:

- In Windows, `conda` commands must be run in the "Anaconda Prompt", located within the "Anaconda" folder in your Start Menu;
- In Unix systems, the `conda` command can be run directly in the Unix Shell (Bash).

We provide an example of how to upgrade a package in the next section; more information can be obtained from `https://conda.io/docs/test-drive.html#managing-conda`.

[1] Available at `https://conda.io/docs/install/quick.html`

1.1 The Spyder IDE

We will use the Spyder Interactive Development Environment (IDE), which is included in Anaconda; it should be available at the Start Menu (in Windows machines) after installing Anaconda. By default, the Spyder IDE contains a file editor on the left-hand side, an IPython ("Interactive Python") console on the bottom right, and a "Variable Explorer"/"File Explorer"/"Help" pane on the top right. An excellent tutorial of Spyder's main functionalities is available inside Spyder itself, via the "Help" menu - "Spyder Tutorial"; make sure to have a look at it.

You will often be required to download and use special functions and data files in your codes; make sure these files are "visible" to your current Python program by having the Spyder working directory set to the same folder where they are located. Remember to always check if the current Spyder working directory, displayed on the top right of the Spyder screen, is set to the correct location; if not, you can change directory via the directory bar itself or via the "File Explorer" pane.

1.1.1 Updating Spyder

A new version of the Spyder IDE is released every few months, and we recommend keeping your local Spyder installation always up-to-date; this can be done with `conda` by running the command

```
conda update spyder
```

inside the "Anaconda Prompt" for Windows machines, or inside the shell for Unix machines. We assume you will be running the most recent version of Spyder, so be sure to update Spyder immediately after installing the Anaconda Python Distribution in your personal computer. You don't have to worry with this when using University computers.

1.1.2 Configuring Spyder

Before starting the module, we would like to ask you to adjust some options in Spyder for a more interactive and user-friendly experience. In Spyder, go to "Tools" - "Preferences" and:

- In "IPython Console" - "Graphics", select *Automatic* as your default graphics backend; this should allow new figures to pop up as new windows (instead of being embedded in the IPython console), allowing you to manually inspect them using the zoom tools;
- In "Run" - "General Settings", make sure that *Clear all variables before execution* is selected (ticked); as the description suggests, this functionality will delete previously created variables from your workspace when you run a file, so you can be sure that only variables you have explicitly created for this file will be using your memory;
- While still in "Preferences", we also suggest going to "Help" and enabling the automatic connections to be made for both the Editor and for the IPython console; that will enable the "Help" pane (located right above the console) to automatically open the documentation for a particular function you start to write either at the Editor or at the console. The functions documentation usually include a short description of the function, the types of input and output variables it accepts and returns, and sometimes a few examples of its use.

1.2 The `pyaudio` module

By default, neither the Python language library nor the Anaconda Python Distribution contain functionalities for playing and recording sound files directly from a Python script or the IPython console. In order to do that, we need to use an external module called `pyaudio`; it provides Python bindings for PortAudio, a very popular cross-platform audio I/O library.

`pyaudio` is already installed in the University computers, so you shouldn't have to worry with it. In order to install `pyaudio` in your own computer, we need to use `pip`, a more generic Python package manager - i.e. not limited to the Anaconda environment; please run

```
pip install pyaudio
```

inside the "Anaconda Prompt" for Windows machines - or inside the shell for Unix machines - to install `pyaudio`. You should only need to do this once.

1.2.1 The `duplexAudio` module

The `pyaudio` package, while very powerful, is not very straightforward to use. Hence, we will provide a Python module called `duplexAudio`, developed by ISVR researchers, that implements a more user-friendly set of functions around `pyaudio`.

Once downloaded, we suggest permanently moving the `duplexAudio.py` file to a folder of your preference; in order to use `duplexAudio` within Spyder, we must add the folder location to the `PYTHONPATH`:

- Go to "Tools" - "PYTHONPATH manager", select "Add path", and select the folder where `duplexAudio` is located;
- Close the "PYTHONPATH manager" and restart Spyder.

This concludes the setting up process; you should now be able to use `duplexAudio` in your Python scripts, as will be demonstrated below.

2 Introduction to Python

Python is a general-purpose programming language, with a large number of modules available for scientific computing. This tutorial will cover some basic programming concepts, such as: creating arrays of numeric data, manipulating and plotting data.

You will receive a script containing sections of code, but some parts will be missing; use the IPython console to interact with the available data and fill in the gaps with your own code.

2.1 Creating and Plotting Signals

We will start by creating and plotting some signals. We will use the Numpy package to create and manipulate N-dimensional arrays of sampled data, and the Pyplot framework (a part of the Matplotlib package) to plot data; see the reference sheet for general instructions on the Python language and these packages.

Signals are generally obtained from experimental data as a series of values sampled from a sensor at equispaced time instants; each sample is stored as a single number, and the collection of samples is generally stored as a one-dimensional array (i.e. a "vector"). Most signals are a function of time, and hence we must start by creating and plotting the time base: an array containing the time instants at which the signal values are sampled or generated.

```
# import the necessary modules
import numpy as np
import matplotlib.pyplot as plt

# define a sampling frequency (in Hz)
fs = 44100

# sampling interval (in seconds)
```

```
dt = 1/fs

# total length for the time base (in seconds)
T_total = 0.5

# create a time base using 'np.linspace' command
# -> syntax: np.linspace(start, end, number_of_points)
t = np.linspace(0, T_total-dt, fs*T_total)

# open a new figure and plot the time base
plt.figure()
plt.plot(t)
plt.title("Time base vector")
plt.xlabel("samples")
plt.ylabel("time")
```

We can make a few checks to confirm this time base is indeed what we expected it to be:

- We can verify how many samples this time base has by typing `t.shape` in the IPython console: this command tells us how many elements the array `t` has in each of its dimensions. How many dimensions and how many elements in each dimension do you believe `t` must have?
- Arrays generated through `np.linspace` have its elements evenly spaced, so the difference between the elements at any two consecutive indices `t[i]` and `t[i+1]` must be identical to the sampling interval; verify this is indeed the case. Note that Python uses zero-based indices - i.e. the first element in an array or sequence has index 0;
- In Python, we can use negative indices to "wrap around" and read the array starting from the last elements: the last entry in an array can be accessed with the index `-1`, the second-to-last entry with the index `-2`, and so on. What should be the "standard" index for the last element of the array `t`? What should its content be?

Let us now use this time base and synthesize a signal containing a 500 Hz sine wave:

```
# sine wave frequency
f0 = 500

# create and plot the sine wave signal
x = np.sin(2*np.pi*f0*t)

plt.figure()
plt.plot(t, x)
```

- Add your own title and label the axes of this new figure!

You probably cannot see much in this figure as it is; use the zoom and the pan buttons to inspect the plot in more detail, and press the "home" button to reset to the original view. Try to zoom in around one of the peaks; you should note that the graph appears to be continuous, but it is instead made of a series of straight lines connecting two adjacent samples. This fact can be made evident by plotting the same signal with a marker at the samples locations, and not using a line to connect them; let's use red circles (with the extra argument "`ro`") as markers:

```
plt.plot(t, x, "ro")
```

Note that unless you explicitly create a new Figure, `pyplot` will add the new plot to the currently active Figure. You should now be able to visualise the individual samples created by your code; each point represents one sample of the `x` array.

Now, create and plot:

- a sine wave of a different frequency;
- a sine wave with a different amplitude;
- a delayed sine wave;

2.2 Playing Signals with `pyaudio` / `duplexAudio`

Let us now use the `duplexAudio` module to reproduce this signal using the computer sound card and headphones; this module uses the `pyaudio` module internally, and it is covered in more details in the Reference Sheet. For now, simply use the code provided below.

IMPORTANT: Note that this audio file stream is going directly to the audio output, and thus it might be **LOUD**! Please remember to turn down the volume of your sound reproduction system (headphones or computer speakers) before playing any audio file, and then turn the volume up until you reach a comfortable listening level!

```
# no need to import 'pyaudio' explicitly;
# 'duplexAudio' does that internally
from duplexAudio import duplexAudio

# Scale the test signal to avoid clipping and numerical errors
gain = 0.5

# Set the block length for replaying the audio signal.
blockLength = 512

# Use the provided library function to play back the file
# using defaults for arguments
#
# Recommended default APIs:
# --> in Windows: audioApi = "MME"
# --> in Linux/Ubuntu:  audioApi = "ALSA"
# --> in Mac: audioApi = "CoreAudio"
duplexAudio(x*gain, fs, blockLength, audioApi="MME")
```

By now, you should have a code that creates, plots and plays back a sine wave. Save your file, and try running it with a few different parameters; for example:

- Plot and play sine waves at frequencies 250 Hz, 500 Hz and 750 Hz; what do you expect will happen?
- Change the sampling frequency to 1000 Hz and repeat the above exercise; what do you expect will happen?

2.3 Summing Signals - Fourier Synthesis

A more complicated periodic signal can be built from a sum of sine and cosine waves using Fourier synthesis; this next exercise will cover how to approximate a sawtooth wave with a partial Fourier sum. A discrete Fourier sine series can be built as the sum of the first M harmonics:

$$x[n] = \sum_{m=0}^{M-1} b_m \sin(2\pi m f_0 t[n]), \tag{1}$$

where f_0 is the fundamental frequency. The function `bm_saw(M)` shown below returns an array containing the first M coefficients for the Fourier sine series of a sawtooth wave:

```
def bm_saw(M):
    m = np.arange(M)
    bm = -2*((-1)**m)/(np.pi*m)
    bm[0] = 0

    return bm
```

Repeating tasks, such as building a Fourier series, are well suited for using a `for`-loop: a code section that repeats itself a fixed number of times while iterating over a sequence; see the Reference Sheet for the basic details of `for`-loops. The code below creates an empty array and sums a single Fourier sine component to it; modify this code so it will sum all terms based on the length of the array of coefficients `bn`.

```
def fourier_sine(f0, bn, t):
    x = np.zeros(t.shape[0])

    for n, b in enumerate(bn):
        x += b*np.sin(2*np.pi*n*f0*t)

    return x

fs = 44100
dt = 1/fs
T_max = 0.4
f0 = 1000

t = np.linspace(0, T_max-dt, fs*T_max)

plt.figure()
for N_saw in range(1, 5):
    saw_sine = fourier_sine(f0, bm_saw(N_saw), t)
    plt.plot(t[:2*fs//f0], saw_sine[:2*fs//f0])

    duplexAudio(saw_sine, fs, 1024, audioApi='ALSA')
```

2.4 Reading `.wav` files

When performing experiments, the recorded data is often stored as a `.wav` file; we will now proceed to read one such file and analyse it. Make sure the file `file1.wav` is in your current working directory and execute the following code:

```
import scipy.io.wavfile as wavio
from wavToFloat import wavToFloat

# obtain the sampling frequency and the bitstream
fs, x_raw = wavio.read("file1.wav")

# Convert the samples to floating-point numbers
```

```
x_wav = wavToFloat(x_raw)

# create the time base
t = np.linspace(0, (x_wav.shape[0]-1)/fs, x_wav.shape[0])

plt.figure()
plt.plot(t, x_wav, "b-")
plt.title("file1")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
```

- Visually inspect the signal by using the zoom tools; what do you think this signal represents?
- Use the code for playing signals with `duplexAudio` and listen to this signal; did you guess correctly?

2.5 Array Indices and Array Content

As previously discussed, arrays and sequences are indexed; one can then read a portion of an array using slicing, denoted by square brackets after an array name. We write

```
x[i:j]
```

in order to obtain all elements between indices `i` (included) and `j` (not included); this operation will return an array with `j-i` elements. The Reference Sheet contains more information on slicing operations; make sure you understand it well.

- The first segment of the `file1.wav` signal contains a few milliseconds of background noise. From visual inspection, estimate the temporal length of this initial segment; how many samples does this interval corresponds to? Open a new Figure and plot this initial segment only using the slicing notation;
- The second segment of this signal corresponds to an almost periodic oscillating signal; what is the approximate period of the oscillations in seconds? And in samples? Obtain a range of indices containing a single period of the oscillations and plot it;

One very important distinction to keep in mind is the difference between the *index* of a given element in an array - i.e. its position in the array - and the *content* of this element - i.e. the value that is stored at that position. The following exercise explores this concept.

- Identify the first period of the oscillations and plot it against time in a separate Figure;
- Use the `np.max()` function to find the amplitude of the highest peak in this signal segment, and use the function `np.argwhere()` to identify the sample where the maximum occurs;
- What is the time instant (in seconds) where the maximum amplitude occurs?

2.6 Conditional Statements - `if`

Write a function `traffic_light(load)` that takes a floating point number load. The function should return the string:

- "green" for values of load below 0.7;
- "amber" for values of load equal to or greater than 0.7 but smaller than 0.9;

- "red" for values of load equal to 0.9 or greater than 0.9.

Example:

```
In [ ]: traffic_light(0.5)
Out[ ]: 'green'
```

Function prototype:

```
def traffic_light(load):
    """Concise one-line sentence describing the function.

    Complete function documentation, which can contain multiple
    paragraphs. This text should appear on your 'Help' pane when
    you write the function in the Editor or console and help you
    use it correctly."""

    return None
```

2.7 Advanced Exercise: Estimating Heart Rate

This next exercise should cover a little bit of all the topics discussed so far; we expect that by the end of this Tutorial session, you should have the necessary know-how and tools to solve it at your disposal.

The file `ecg.wav` contains approximately 10 seconds of an ECG signal recorded from a patient. Your tasks are:

- load the file and visually inspect the signal;
- create a Numpy array with the data corresponding to the first 5 peaks;
- use a 'for' loop to read the array in segments corresponding to each peak;
- locate the peak amplitude of each segment, and find its corresponding sample value and time instant;
- plot the peak locations on top of the original signal and check if you located them correctly;
- calculate the average heart rate (in beats per minute - BPM) for this signal based on the time interval between the peaks.

Numpy has some in-built functions that might help with some of these tasks; do search for them. Pay extra attention to how the indices in each segment correspond to the indices in the original array; the same is valid for the time instant of each peak.

The heart rate we obtained is approximately 93.7 beats per minute.