University of Southampton
Institute of Sound and Vibration Research
MSc Sound and Vibration Studies
Python Quick Reference Sheet

September 27, 2017

Contents

This document contains a selection of Python commands to be typed and executed at the IPython console; they should demonstrate the functionalities of the main commands and variable types you will probably use when programming in Python. Before running the commands, we would like you to think what will be the console output, and to explain why you think so based on your knowledge of the Python programming language. You don't have to remember all the details presented in this sheet, but you might want to go through them all at least once.

You are also encouraged to find resources and learn by yourself whenever necessary; there is a plethora of resources on Python programming on the internet, and we suggest a few references at the end of this reference sheet. This material is heavily inspired by the *Scipy Lecture Notes* (see References).

1 Variable Types

Variables are defined by assigning a value to a name using the equality ("=") operator. Think of the variable name as a "label" associated with a particular location in the computer memory where the value is stored; pre-existing names can be redefined inside the code, and multiple names can reference the same variable.

```
a = 3
type(a)
print(a)

b = 2*a
type(b)
b

a*b
a**b

# variable type can change during execution
b = "hello"
type(b)
print(b)

b + b
3*b
```

2 Numerical Variable Types

Numerical variables can be of type `int` (integer), `float` (floating-point number) or `complex` (complex number).

```
a = 4
type(a)

b = 2.1
type(b)

# type conversion
c = float(1)
type(c)

# conversion to integer is not the same as rounding!
int(2.1)
round(2.1)

int(2.9)
round(2.9)

# complex type (real plus imaginary parts)
d = 1.5 + 0.5j
type(d)
d.real
d.imag
```

NOTE: Integer division has different behaviours in Python 2 (where `int / int = int`) and Python 3 (where `int / int = float`); pay attention to the variable types, and think what is the behaviour you want from your code. If in doubt, use explicit type conversion.

The floor division (`//`) and the modulo (`%`) operators are connected through the expression `x == (x//y)*y + (x%y)`.

```
a = 3
b = 2

# In Python 2 , this is equal to 1; in Python 3, this is equal to 1.5
a / b

# using explicit type conversion to force float division
a / float(b)
3/2.

# floor division
a//b

# If one operand is float, result is also float!
float(a)//b

# modulo operator
a % b
```

3 Lists and Tuples

Lists are ordered collection of objects, which can be of different types. Indexing starts at zero, and negative indexes allow counting backwards from the end of the list.

```
L = ["red", "blue", "green", "black", 1001.099]

type(L)
len(L)

L[1]
L[0]

L[-1]
L[-4]

# list elements can be changed
L[3] = 1000
L
```

Slicing results in sublists of regularly-spaced elements: `[start:stop]` returns all elements with indexes `start <= i < stop` (i.e. `i` ranges from `start` to `stop-1`, so that `stop-start` elements are returned). One can also add a step size using the synthax `[start:stop:step]`.

```
L[0:3]

L[2:3]

L[0:4:2]
```

```
L[:]

L[::-1]
```

Tuples are similar constructs to lists, but are immutable and generally use round brackets.

```
t3 = ("John", "Johanna", 999)

type(t3)

t3[2]

# tuples cannot be changed
t3[1] = 10

# single-element and empty tuples are valid
u = (10,)
v = ()

# tuples can be simultaneously unpacked
t0, t1, t2 = t3

type(t0)
type(t2)
```

4 Strings

Strings are "lists of characters", and are indexed just like lists.

```
a = "hello, world!"
a[3:6]

a[2:10:2]

a[::3]
```

String formatting allow the creation of strings using previously defined variables.

```
a = 1
b = 0.1
name = "John"
print("An integer: {}; a float: {}; another string: {}".format(a, b, name))

# use three decimal places for float inside string
pi = 3.14159265359
"The number pi can be approximated as {:.3f}".format(pi)

# exponential notation with two decimal places
"The number 123456 can be approximated as {:.2e}".format(123456)
```

5 Control Flow

5.1 `if` tests

`if` tests allow for conditional execution of code when a given condition tests true; alternative tests can be added using `elif`, and exceptions to all tests are dealt with by using `else`.

Note that blocks are denoted by indentation; this can be achieved with the `Tab` key, or with a consistent number of whitespaces (normally four; Spyder automatically "translates" `Tabs` to four whitespaces).

```
condition = True
if condition:
    print("It's true!")

# one equal sign means variable assignment;
# two equal signs means comparison!
a = 5

if (2+2 == a):
    print("It's four!")
elif a < 0:
    print("It's negative!")
elif (a < 4) and (a > 0):                # conditions can be concatenated
    print("It's less than four, but positive!")
else:
    print("It's more than four!")
```

5.2 `for` loops

`for` loops allow the repeated execution of a given piece of code while iterating over a counter or sequence.

```
# can iterate over elements of a sequence
for word in ("cool", "easy", "fun"):
    print("Audio is {}".format(word))

# The iterator 'range' also follows the (start, stop, step) notation
for i in range(4, 10, 2):
    print(i)
```

When looping over the entries in a list, it is important to keep in mind the differences between the *index* of a list entry and the *content* of the list entry; using the built-in function `enumerate` might help with that:

```
x = [2, 4, 6, 8, 10]

# 'i' is the index, 'xi' is the content
for i, xi in enumerate(x):
    print("The {}-th entry in x is {}".format(i, xi))
```

6 Functions

Functions allow the encapsulation and abstraction of a task, and thus stimulates reusing code instead of rewriting code. Python function calls use parentheses (round brackets) for the function arguments;

functions return `None` by default, unless explicitly stated otherwise.

```
def double_it(x):
    """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 x * 2

double_it?

double_it(3)

double_it()
```

Python functions can have default argument values for when a function is called without one or more of its arguments.

```
def power_it(x = 2, y = 2):
    return x**y

power_it(3, 3)

power_it(3)

power_it()
```

7 Numpy

Numpy is the "numerical Python" module; it contains its own homogeneous, multi-dimensional array type, which is useful for fast numerical calculations. One-dimensional arrays are equivalent to vectors (in the mathematical sense), and its elements are indexed using a single index; two-dimensional arrays are equivalent to matrices (in the mathematical sense), and its elements are indexed using two indices; and etc. Numpy must be imported before being used.

```
import numpy as np

# notice lists (and lists of lists) are used as input
# arguments for creating arrays
a = np.array([0, 1, 2, 3])              # 1D array
b = np.array([[0, 1, 2], [3, 4, 5]])    # 2D array (2x3)

a
b

# 'shape' of array: number of elements in each dimension
a.shape
b.shape

# 'size' of array: total number of elements
a.size
b.size
```

```
# check data type inside array
a.dtype

# arrays use same indexing as lists
a[1]
b[0, 1]

b[1]

# slicing works independently in each dimension
b[0, :]
b[:, 0]
```

Some common functions for creating arrays are shown below.

```
# arange: (start, end (not included), step_size)
b = np.arange(1, 9, 2)
b

# linspace: (start, end, number_of_points)
# endpoint can be included (True - default behaviour) or not (False)
c1 = np.linspace(0, 1, 6)
c1

c2 = np.linspace(0, 1, 6, endpoint=False)
c2

# zeros: creates an array full of zeros
# note that the tuple '(2, 3)' is considered a single input argument!
z = np.zeros((2, 3))
z

# sometimes it is worth creating a matrix of 'complex zeros'
z_complex = np.zeros((3, 3), 'complex')
z_complex.dtype
z_complex

# ones: similar behaviour as 'zeros'
o = np.ones((3, 5))
o

# create array of random numbers (normal/Gaussian distribution)
r = np.random.randn(10)
r
```

Arrays can be read and modified using their indexes; each dimension requires a separate index.

```
array_shape = (4, 4)
A = np.zeros(array_shape)
A

A[1, 3] = np.pi
A

b = np.arange(5, 9)
```

```
A[2, :] = b
A

d = np.array([1, 2, 3])
# sum with existing content (instead of overwriting)
A[1:, 0] += d
A
```

Note that `np.arange` is NOT recommended for non-integer steps, because it is prone to errors due to numerical precision. If non-integer intervals are required, we highly recommend using `np.linspace` instead.

```
# How many elements should this array contain?
# What should be their values?
#   --> Remember: arange should NOT include the endpoint!
a = np.arange(0, 1, 1./10)
a
a.size
a[-1]

# How many elements should this array contain?
# What should be their values?
b = np.arange(0, 1, 1./49)
b
b.size
b[-1]

c = np.linspace(0, 1, 10, endpoint=True)
c
c.size
c[-1]

d = np.linspace(0, 1, 49, endpoint=True)
d
d.size
d[-1]
```

Array operations are almost always performed element-wise:

```
x = np.arange(5)

x+2

x*3

x**3

np.sqrt(x)

x*x

np.exp(x)

# base-10 logarithm
np.log10(x)
```

Some operations are not performed element-wise; amongst those, we include the dot product (matrix multiplication when applied to 2D arrays, and vector inner product when applied to 1D arrays) and the cross product (also known as outer product). The performance of these operations on N-D arrays is more complex, be careful with unexpected results.

```
x = np.arange(3)
y = np.arange(4, 7)
A = np.arange(9).reshape((3, 3))

# dot product / inner product
np.dot(x, y)

np.dot(A, x)

# on newer Numpy versions, '@' can be used for
# matrix product (in the mathematical sense)
np.dot(A, x) == A @ x

# cross product / outer product
np.outer(x, y)
```

Numpy arrays can also be concatenated, as long as their respective shapes match; there are other methods for concatenating arrays, try to research the others.

```
x = np.array([[0., 0.1, 0.2], [0.3, 0.4, 0.5]])
y = np.array([[1., 1.1, 1.2], [1.3, 1.4, 1.5]])

x.shape
y.shape

# function argument is a tuple '(x, y)'
z1 = np.concatenate((x, y))                    # 1st axis by default
z1.shape
z1

z2 = np.concatenate((x, y), axis=1)
z2.shape
z2
```

8 Matplotlib

The Matplotlib module is a very powerful module for plotting, but can be a bit cumbersome to use; we recommend using the `pyplot` interface, which adopts a `MATLAB`-like style and simplifies some of the most common commands. It must be imported before being used.

The Scipy Lectures website has a very good "Quick Reference" guide for Matplotlib commands; see References.

```
import matplotlib.pyplot as plt
import numpy as np

# 1D data
x = np.linspace(-np.pi, np.pi, 256)
y1 = np.cos(x)
y2 = np.sin(x)
```

```
plt.figure()
plt.plot(x, y1, label="y1")
plt.plot(x, y2, 'ro', label="y2")

plt.xlabel("x label")
plt.ylabel("y label")
plt.title("Simple Plot")
plt.legend()
plt.savefig("fig_name.png")
```

9 Playing and Recording Sound Files

We will now instruct you on how to play and record sound files in your computer using Python. We will use the `pyaudio` module, but we will interact with it using the wrapper functions available in the `duplexAudio` module. First, we need to import the required modules:

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

# functions to create periodic signals
from scipy.signal import square, sawtooth

# functions to read and write .wav files
import scipy.io.wavfile as wavio

import numpy as np
```

9.1 Reading and Reproducing `.wav` Files

First, let us inspect the contents of a Wave file, as used by most media players. This is an audio file format standard used to store an audio bitstream, similarly to the method used to store music in CDs; such files are usually recognised by their `.wav` extension. Wave files store each sample as a signed `N_bits` integer (i.e. the first bit contains the sign and is not used to store amplitude data); for example, CD audio uses 16 bits per sample.

The `scipy.io.wavfile` module allows for reading and writing Wave files and obtaining their bitstream. However, audio data is often manipulated using floating-point representation (such as Numpy arrays), and it is then necessary to convert the data type to Python's `float` type; for this, we can use the `wavToFloat` module.

Before running this next example, make sure the `.wav` file is located inside the current Spyder working directory; the file should be visible to you in the "File Explorer" pane.

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

# Convert the samples from int to float
x = wavToFloat(xRaw)

# Plot the waveform
t = np.linspace(0, (x.shape[0]-1)/fs, x.shape[0])
```

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

Explore the waveform by zooming in and out using the magnifying glass and the quad arrow buttons in the Figure window; try zooming in around any particular features you believe might be important.

Now, let us play to the Wave file recording:

```
# 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")
```

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!

9.2 Reproducing and Recording Signals from within Python

Let us now create some signals and listen to them:

```
# create a vector of time samples
T0 = 1.             # signal length [s]
dt = 1/fs
t = np.linspace(0, T0-dt, T0*fs)

f0 = 440            # frequency of A4 note [Hz]

# *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
# create, plot and play a sine wave
x_sin = 0.25*np.sin(2*np.pi*f0*t)

plt.figure()
plt.plot(t[:500], x_sin[:500])  # plot the first 500 samples
plt.title("Sine Wave")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.grid()

duplexAudio(x_sin, fs, blockLength, audioApi="MME")
```

```
# *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
# create, plot and play a square waves
x_square05 = 0.1*square(2*np.pi*f0*t, duty=0.5)

plt.figure()
plt.plot(t[:500], x_square05[:500])
plt.title("Square Wave")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")

duplexAudio(x_square05, fs, blockLength, audioApi="MME")

# *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
# create, plot and play a sawtooth wave
x_saw = 0.1*sawtooth(2*np.pi*f0*t)

plt.figure()
plt.plot(t[:500], x_saw[:500])
plt.title("Sawtooth Wave")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")

duplexAudio(x_saw, fs, blockLength, audioApi="MME")
```

Finally, `pyaudio/duplexAudio` allows for simultanous playback and recording of audio data; this next example will demonstrate how this can be done by recording a few seconds of data, plotting the recorded waveform, and then playing it back to you.

Note that the file being played back and the recorded file need not have the same length; hence, if you don't want to play anything while recording, you only need to "play" a Numpy array containing a few zeros.

```
# create array with a single zero
x_silence = np.zeros(1)

# record 2 s of data and save in array 'y'
length = 2
y = duplexAudio(x_silence, fs, blockLength, recordLength=length,
                audioApi="MME")

# create vector of time samples
t2 = np.linspace(0, length-dt, length*fs)

# plot vector 'y'
plt.figure()
plt.plot(t2, y)
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")

# play the recorded file 'y'
duplexAudio(y, fs, blockLength, audioApi="MME")
```

10 References

Some useful references:

- Prof. Hans Fangohr's *"Python for computational science and engineering"* textbook and slides

  `http://www.southampton.ac.uk/~fangohr/teaching/index.html`

- Scipy Lecture Notes:

  `http://www.scipy-lectures.org/`

- Software Carpentry *"Programming with Python"* lessons:

  `http://software-carpentry.org/lessons/`

- Python's Official Tutorial:

  `https://docs.python.org/3/tutorial/`

- Matplotlib's Pyplot Tutorial:

  `https://matplotlib.org/users/pyplot_tutorial.html`

- *"Best Practices for Scientific Computing"* paper:

  `http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745`

- *"Good Enough Practices in Scientific Computing"* paper:

  `http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510`

Please note that this revision does not cover everything about programming with Python. There is a wide range of tutorials and sources about Python on the internet, and you are expected to search for more information and learn new things by yourself.

We recommend reading the articles *"Best Practices for Scientific Computing"* and *"Good Enough Practices in Scientific Computing"*, written by some of the colaborators of the Software Carpentry team. While we acknowledge that following the series of "best practices" to the letter can be very demanding, it is important to at least be aware of what are considered best practices and why; meanwhile, the "good enough" practices are more straightforward to implement and within immediate reach of a researcher working by him/herself.