Applications for Digital Design and Signal Processing
Session 4

Dan Boschen

Course Outline

Session	Topics
1	Course Intro: Python, Spyder and Jupyter
2	Core Python
3	Core Python
4	**Core Python**
5	Python Modules and Packages
6	NumPy
7	NumPy, SciPy
8	Python for Verification, Modelling and Analysis

Session 4 Contents

Goals for this Session: Review of the core Python language: Functions, Comprehensions, Reading/Writing Files, Generators

Functions

Functions

```
>>> def add_these(x, y, z = 5):
...     """
...     docstring
...     """
...     if x < 0:
...         print("x must be positive")
...         return False
...     return x + y + z
```

Functions

default value

parameters

```
>>> def add_these(x, y, z = 5):
...     """
...     docstring
...     """
...     if x < 0:
...         print("x must be positive")
...         return False
...     return x + y + z
```

5, k are used as positional parameters

z is used as a keyword parameter

arguments

```
>>> k = 15
>>> add_these(5, k, z = 12)
```

Argument Use Cases

Parameters as defined:

```
>>> def add_these(x, y, z = 5):
```

Argument use cases (positional vs keyword):

```
>>> k = 15
>>> add_these(5, k, 12)
```
all positional

```
>>> add_these(5, 18, z = 12)
```
kw for 3rd arg

```
>>> add_these(5, y = 15, z = 12)
```
kw for 2nd allowed, 3rd MUST be kw

```
>>> add_these(x = 5, y = 15, z = 12)
```
kw used for 1st arg, so all following args must be kw

Functions

functions are called with ()

functions can be passed into other functions!

```
>>> def do_this(my_other_function):
...     my_other_function()
```

Function nargs and kwargs

```
def my_func(a, b, *nargs, **kwargs):
    # function contents
```

***nargs**: variable number of positional parameters

****kwargs**: variable number of keyword parameters

Function nargs and kwargs

```
def my_func(a, b, *c, **d):
    # function contents
```

c will be a **Tuple**
in function body

d will be a **Dictionary**
in function body

Function nargs and kwargs

```
my_func('1', 2, 3, '4', 5, x = 6, y = '7')
```

```
def my_func(a, b, *c, **d):
    print(a)
    print(b)
    print(c)
    print(d)
```

'1'

2

(3, '4', 5)

{ 'x': 6, 'y': '7'}

Anything following *nargs or **kwargs must be keyword only.

Function Return

```
def my_func(a, b):
    y = a + b
    print(y)
    return y
```

Return is optional, if omitted 'None' is returned.

Anonymous Functions

Lambda (Anonymous) Functions

lambda parameters: expression

```
lambda x,y: x + 5*y
```

Note similarity to:

$$\lambda(x, y) = x + 5y$$

Lambda (Anonymous) Functions

```
lambda x,y: x + 5*y
```

Often used in higher order functions that take functions as an argument, such as **filter, map, and reduce**

filter: Test each item in list and returns true items

map: Apply function to all items in a list

reduce: Apply rolling computation to sequential pairs

Filter, Map, Reduce

Examples

```
>>> x = map(lambda x: x + 't', ['5','3','2','8'])
>>> list(x)
['5t','3t','2t','8t']

>>> x = map(int, ['5','3','2','8'])
>>> list(x)
[5,3,2,8]
```

filter: Test each item in list and returns true items

map: Apply function to all items in a list

reduce: Apply rolling computation to sequential pairs

Comprehensions

List and Set Comprehensions

```
a = [b**2 for b in my_iterable]

a = [b**2 for b in my_iterable if b>5]

a = {b**2 for b in my_iterable}
```

first expression can be any expression (so can have nested list comprehensions)

Dict Comprehensions

Create dictionaries from arbitrary key and value expressions

```
my_list = [1, 5, 7]

{x: x**3 for x in my_list}
```

results in:

```
{1: 1, 5: 125, 7: 343}
```

Dict Comprehensions

Create dictionaries from items method for a dict:

```
my_dict = {1: 1, 5: 125, 7: 343}

{k: v-3 for k,v in my_dict.items()}
```

results in:

```
{1: -2, 5: 122, 7: 340}
```

Reading / Writing Files

Reading / Writing Files

```
f = open(file_name, mode)
```

mode =

'r' read

'w' write

'a' append

Methods:

read(), readline(), readlines(), write(), writelines(), close()

Reading Files

Use "with" to automatically close resource when done

```
with open(filename, 'r') as f:
    for line in f:
        y = do_somthing(line)
```

```
f = open('file.txt', 'r')
for line in f:
    y = do_something(line)
f.close()
```

Writing Files

Use "with" to automatically close resource when done

```
with open('file.txt', 'w') as f:
    for item in my_iterable:
        f.write(f"{item:s}\n")
```

```
f = open('file.txt', 'w')
for item in my_iterable:
    f.write(f"{item:s}\n")
f.close()
```

Reading / Writing Files

Python csv module for reading/writing data in CSV format:

```
import csv
with open('file.txt', newline = '') as f:
  my_reader = csv.reader(f,  delimiter = ',')
      for item in my_reader:
            y = do_somthing(item)

with open('file.txt', 'w', newline = '') as f:
  my_writer = csv.writer(f,  delimiter = ',')
      my_writer.writerows(my_iterable)
```

Namespaces and Scope

Namespaces

Used extensively by Python to organize object references (names)

Python interpreter will search name spaces in the following order:

- Local
- Enclosing
- Global
- Built-in

And raises a `NameError` exception if not found

Scope

```python
a = 5
b = 7
def outer_func():
    print(a)
    y = 12
    def inner_func():
        print(a)
        print(y)
        return b + y + a
    return inner_func()
```

enclosing scope

local scope

global scope

Functions have **read-only** access to enclosing scope in order going out

Generators

Generators

Generators simplify the creation of iterators.

(A generator is an iterator)

Generator Functions are defined by using "yield" in a function definition:

```
def MyGen():
    for count in range(2000):
        yield count
```

When called, MyGen will return a **Generator Iterator**

Generators

```
def list_func(x):
    out = []
    for item in x:
        result = do_this(item)
        out.append(result)
    return out
```

```
def GenFunc(x):
    for item in x:
        result = do_this(item)
        yield result
```

(note PEP8 style guide suggests CapWords convention for naming class objects, which the Generator Function basically is.

Example
Function Definition,

Input: Iterable
Returns: a **List**

Same functionality converted to a
Generator Function,

Input: Iterable
Returns: a **Generator Iterator**

Generators

```
def GenFunc(x):
    for item in x:
        result = do_this(item)
        yield result
        y = do_stuff()
        yield y
        z = more_stuff()
        yield z
```

Generators

Generator Functions simplify making **Iterators**!!!

```
class Gen:
    def __init__(self, start, stop = 20):
        self.index = start - 1
        self.stop = stop
    def __iter__(self):
        return self
    def __next__(self):
        self.index += 1
        if self.index > self.stop:
            raise StopIteration
        return self.index
```

Custom Iterator Class

```
def Gen(start, stop = 20):
    while True:
        start +=1
        yield start - 1
        if start > stop:
            break
```

Generator Function

Coroutines

Coroutines are **consumers** of data

Generators are **producers** of data

Use Yield as an expression to create a coroutine, and use the send() method to pass data in:

```
def GenFunc(x):
    for item in x:
        result = do_this(item)
        y = yield
        print (result + y)
```

Consumer / Producer

```
mygen = GenFunc()
```

```
def GenFunc():
    result = do_this()
    in1 = yield result
      y = do_stuff()
      in2 = yield y
      z = more_stuff()
    in3 = yield z
```

This is what runs when we issue first `next(mygen)`

Consumer / Producer

```
mygen = GenFunc()
```

```
def GenFunc():
    result = do_this()
    in1 = yield result
      y = do_stuff()
      in2 = yield y
      z = more_stuff()
    in3 = yield z
```

This is what runs when we then issue `w = mygen.send(5)`

Zip

zip

zip aggregates iterables from **multiple streams**:

```
>>> a = ['a', 'b', 'c']
>>> b = [7, 'c', 6]
>>> c = [-9, 2, 'd']
```

To a **stream of multiples**, returning an iterator:

```
>>> out = zip(a, b, c)
>>> list(out)
[('a', 7, -9), ('b', 'c', 2), ('c', 6, 'd')]
```

zip

```
>>> list(zip(a, b))
[('a', 7), ('b', 2), ('c', 6)]
```

Easy way to make a dictionary from two iterators representing key and value:

```
>>> dict(zip(a, b))
{'a': 7, 'b':2, 'c':6}
```

To "unzip"

```python
a = ['a', 'b', 'c']
b = [7, 2, 6]
out = zip(a,b)
list(out)
```

zipped:

```
[('a', 7), ('b',2), ('c',6)]
```

```python
unzipped = zip(*out)
list(unzipped)
```

unzipped:

```
[('a', 'b', 'c'), (7, 2, 6)]
```

These are equivalent:

```python
zip(*out)
zip(('a', 7), ('b',2), ('c',6))
```

zip

Useful related idiom with zip:

Divides a long series into groups of n

```
zip(*[iter(series)] * n)
```

This will not work (will repeat each item in iterable):

```
zip(*[iterable] * n)
```

Exceptions

Exceptions

EAFP vs **LBYL**

Exceptions

Error handling is done through exceptions.

Exceptions caught in try blocks and handled in except blocks

```
>>> try:
...     # code that may error
>>> except:
...     # code to execute if error occurs
```

Try / Finally

```
>>> try:
...     # code that may error
>>> except Exception1:
...     # code to execute if error occurs
>>> except Exception2:
...     # code to execute if error occurs
>>> else:
...     # code to execute if no errors occurs
>>> finally:
...     # code that will always be run regardless
```

Exception Example

```
>>> x = 0
>>> y = 5/x
ZeroDivisionError: division by zero

>>> for item in my_list:
...     try:
...           y.append(5 / item)
...     except ZeroDivisionError:
...           # process error and skip long process
...     else:
...           # some long process if no errors occur
...     finally:
...           # clean-up for all items
...
>>> # code continues here
```

Profiling

Profiling

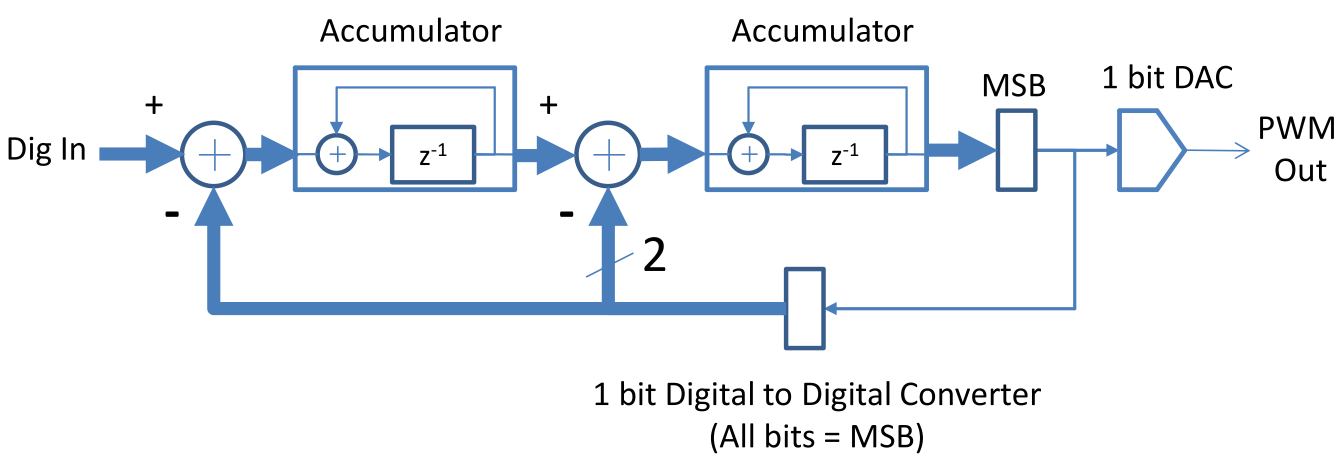
Useful line (%) and cell (%%) magics in Ipython for code profiling:

magic	description
%lprun:	line-by-line profiler <-- need to install line_profiler
%memit:	memory use of single statement <--- need to install memory_profiler
%mprun:	line-by-line memory profiler <--- need to install memory_profiler
%prun:	profiler (% line and %% cell)
%time:	execution time of a single statement
%timeit:	loops over statement multiple times for statistical execution time (% line and %% cell)

Profiling is further detailed in Reference Notebook

Code Example

2nd Order Delta Sigma DAC



HW Assignment

Create a Delta Sigma DAC per the function definition prototype and block diagram on next slides.

function parameters passed in:

input_values: iterable representing desired output level vs time [-5,-5,-5,-5,3,2,....] (over **many** samples output will average to a scaled version of this level, which can change with time)

input_width: single integer representing bit width of input

function returns:

ds_gen: iterator generator (only the values 0 and 1 as the pulse-width-modulated (PWM) output

BONUS CHALLENGE: Make your function accept either an input as a list or a single constant with an optional parameter of total samples to run, with a default duration of 500 samples; for case of a single constant, have it repeat that constant value as input for the total number of samples.

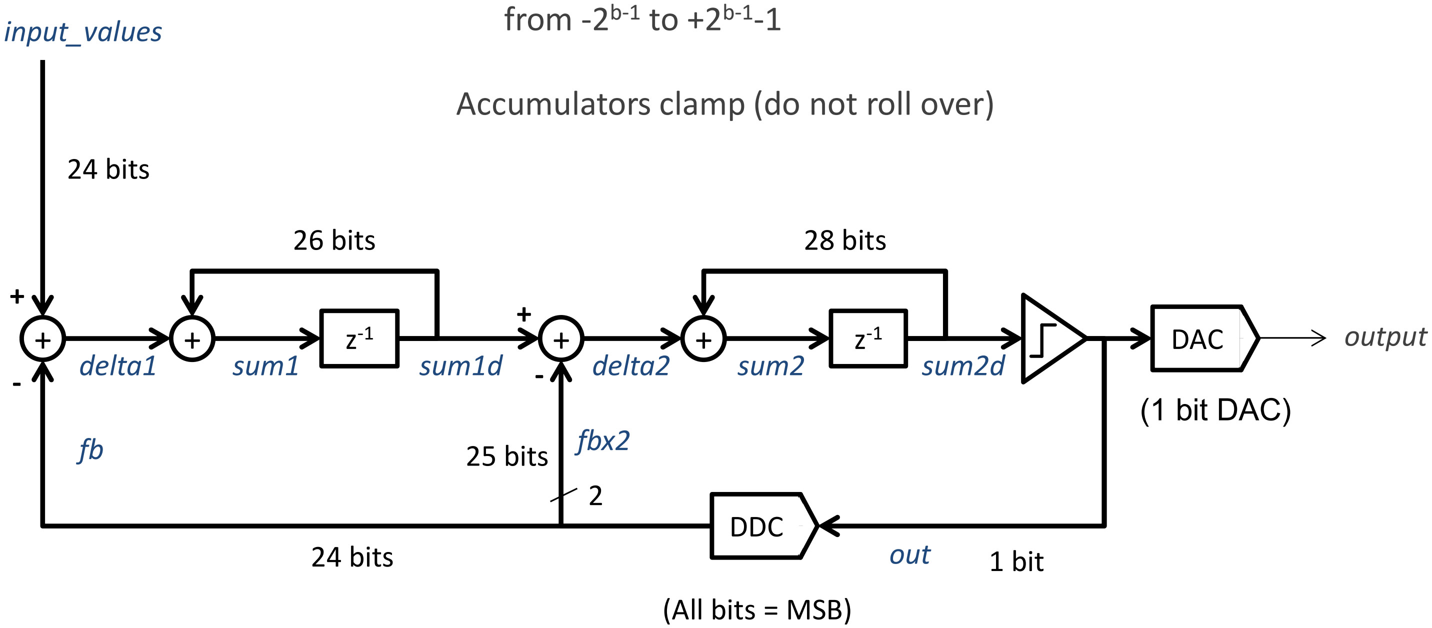
HW Assignment

```
def DsGen(input_values, input_width):
    """
        ....add a doc string....
    """

    # initial values
    ....add code....
    # start for loop:

        # synchronous ops: what happens on rising clk edge
        ....add code....

        # asynchronous ops: what happens in between clks
        ....add code....

        yield ds_gen
```

DAC Implementation

All *values* are represented as signed integers from -2^{b-1} to $+2^{b-1}-1$

Accumulators clamp (do not roll over)

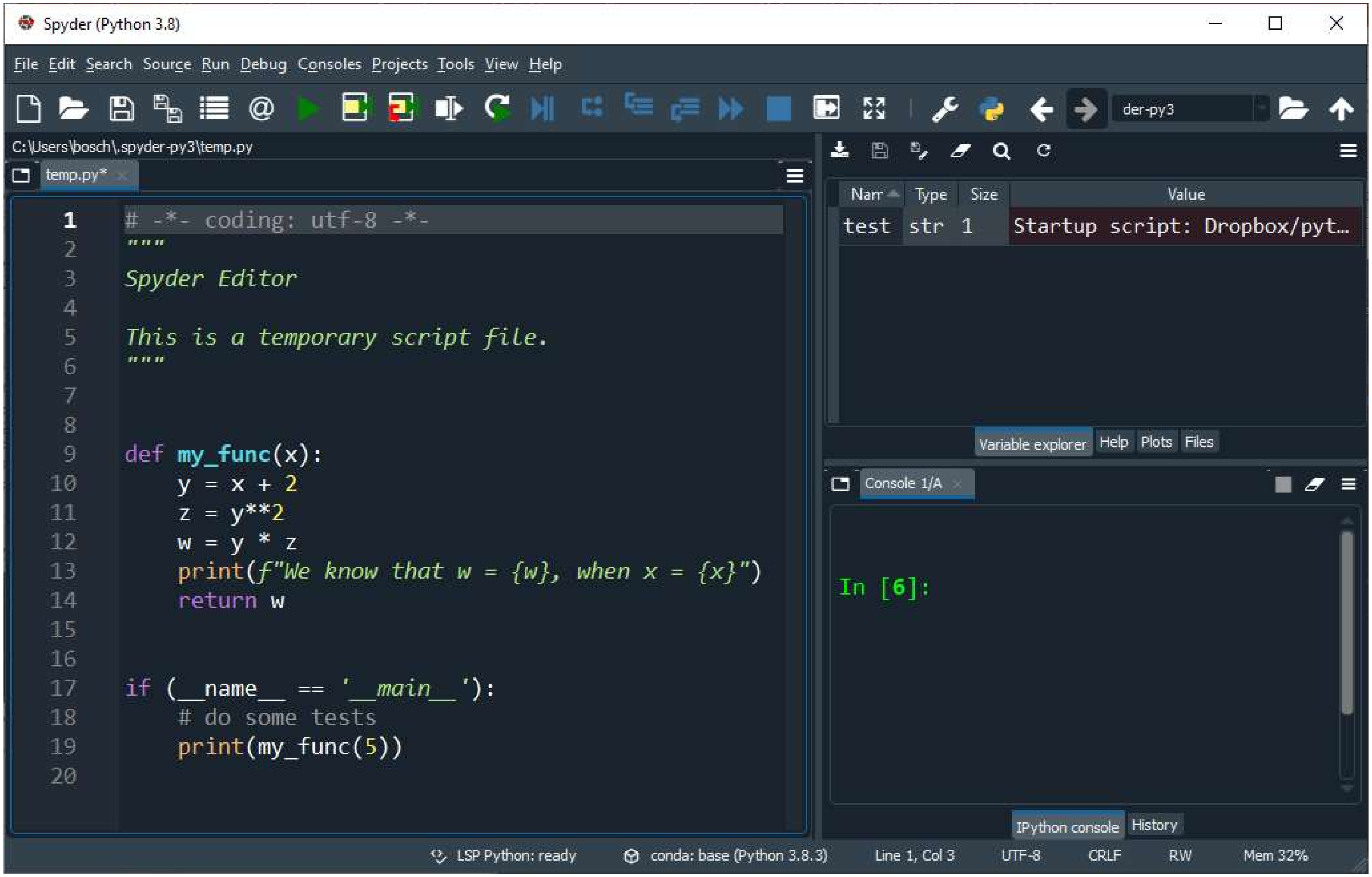


out is 1 bit: out = 0 for sum2d <0; out = 1 for sum2d >= 0
fb is either -2^{b-1} when out is 0, or $+2^{b-1}-1$ when out is 1
fbx2 is approximately 2x fb; specifically use -2^{b} and $+2^{b}-1$

HW Assignment

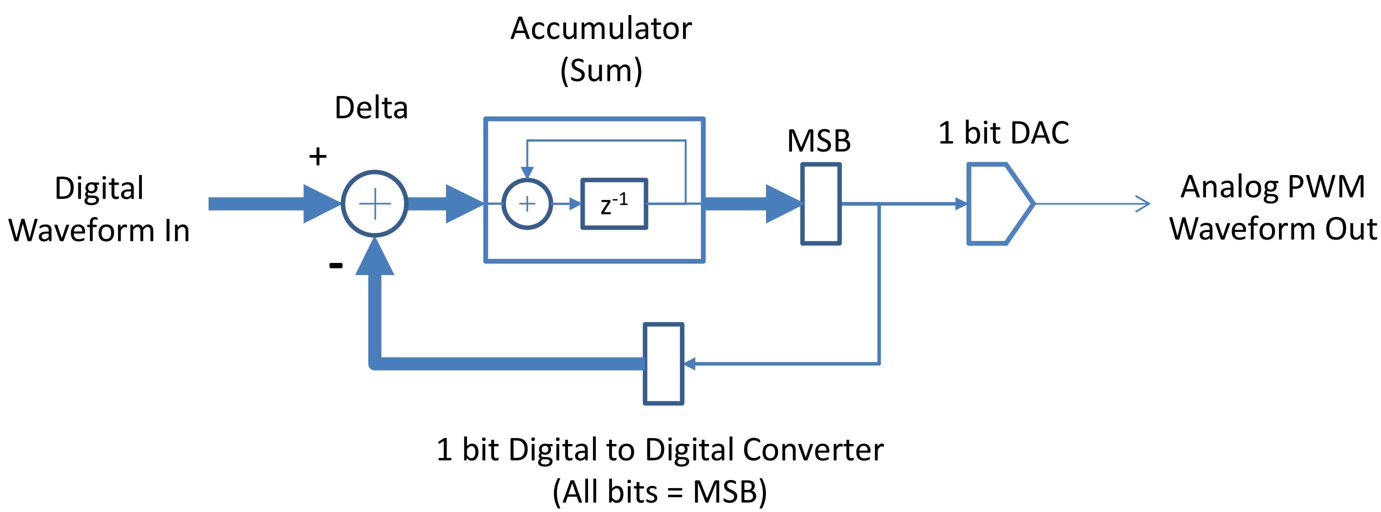
```
while (not_done and its_not_tues):
    result = write_code_using_spyder()
    if (you_dont_get_it):
        review(first_order_implementation_from_mod1)
        review(all_course_slides)
        review(ds_block_diagrams)
        continue
    elif (confused_about_python):
        review(docs.python.org)
        continue
    elif (really_stuck):
        print(curse_words)
        email(boschen@loglin.com, curse_words = None)
        continue
    else:
        download(test_case, dropbox/session4/data)
        compare(result, test_case)
```

Note: Test case for comparing results will be posted to the Session 4 data folder

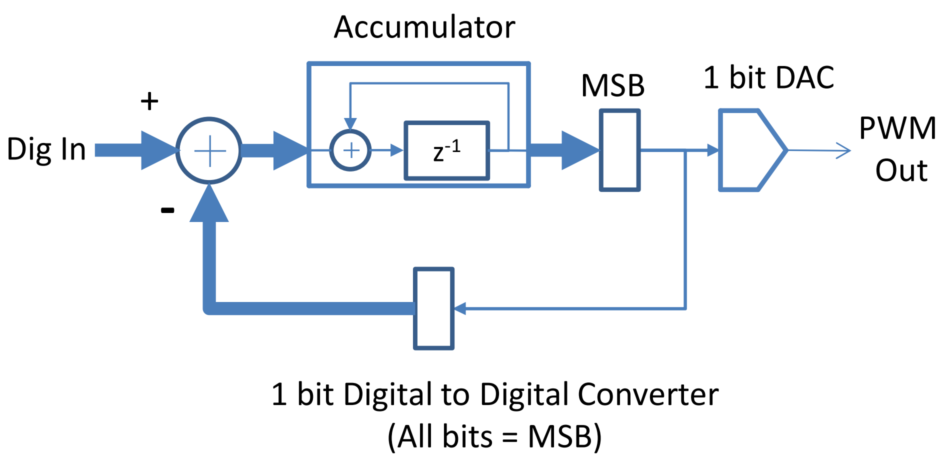

Spyder (Python 3.8)
File Edit Search Source Run Debug Consoles Projects Tools View Help
der-py3
C:\Users\bosch\.spyder-py3\temp.py
temp.py*
-*- coding: utf-8 -*-
"""
Spyder Editor
This is a temporary script file.
"""
def my_func(x):
y = x + 2
z = y**2
w = y * z
print(f"We know that w = {w}, when x = {x}")
return w
if (__name__ == '__main__'):
# do some tests
print(my_func(5))
Name Type Size Value
test str 1 Startup script: Dropbox/pyt...
Variable explorer Help Plots Files
Console 1/A
In [6]:
IPython console History
LSP Python: ready
conda: base (Python 3.8.3)
Line 1, Col 3
UTF-8
CRLF
RW
Mem 32%

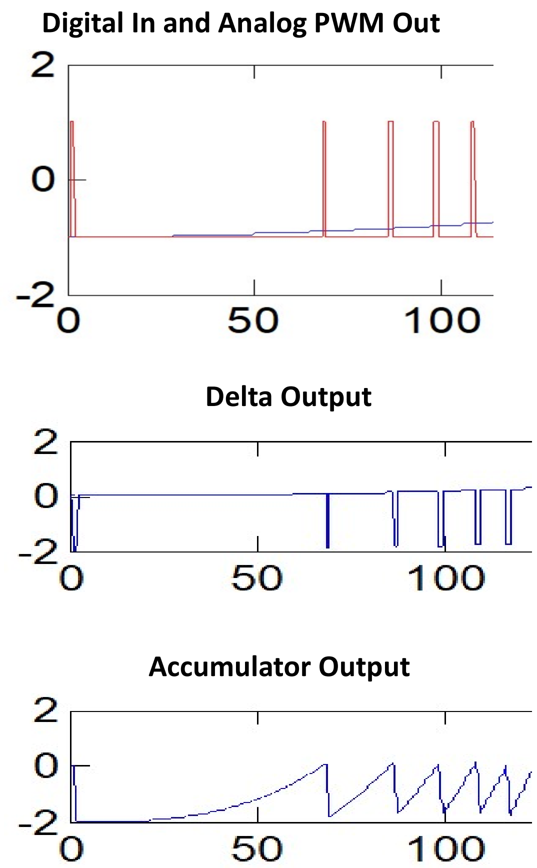
Backup Slides

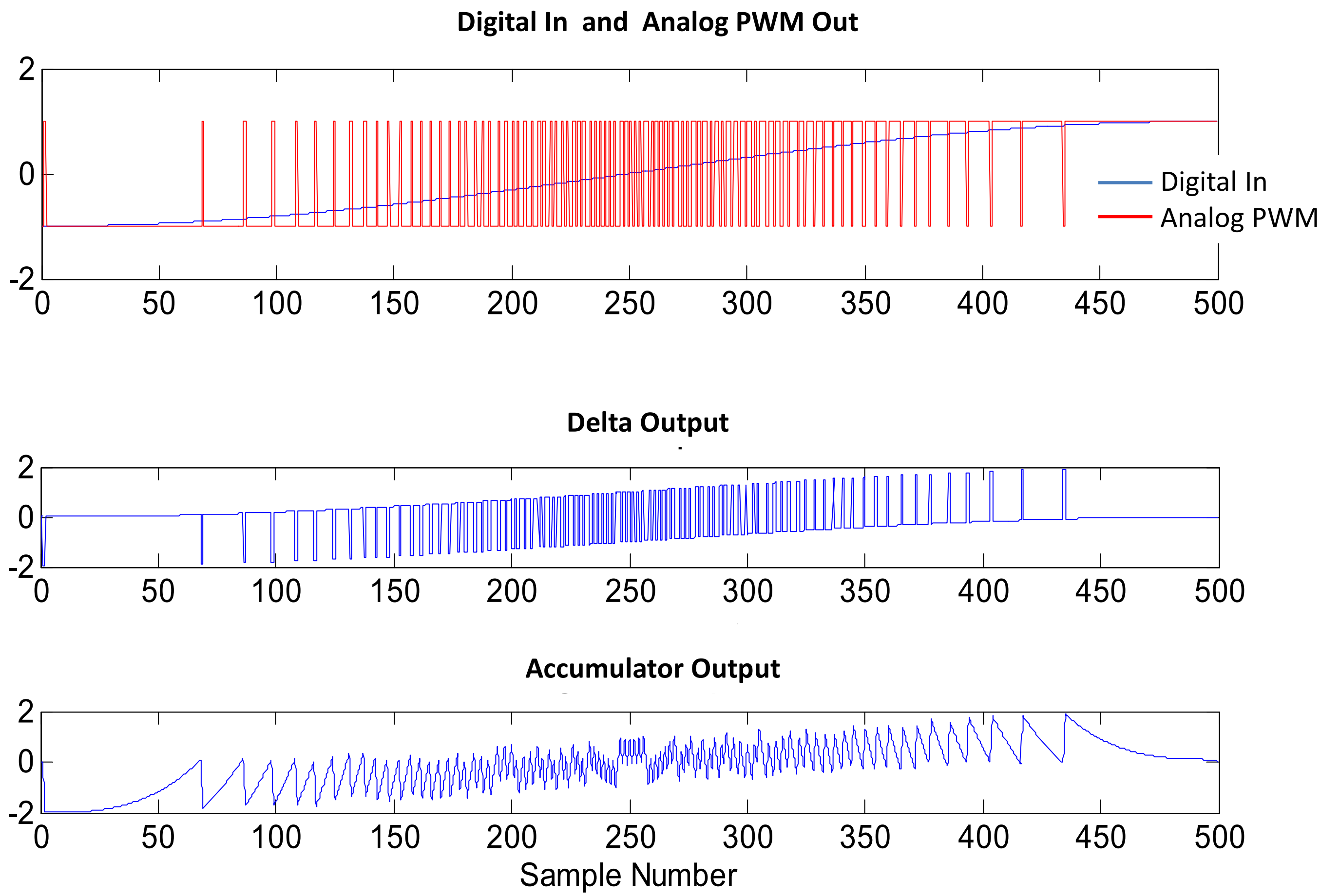
First Order Delta Sigma DAC



Delta Sigma Waveforms






Digital In and Analog PWM Out
2
0
-2
0
50
100
150
200
250
300
350
400
450
500
Digital In
Analog PWM
Delta Output
2
0
-2
0
50
100
150
200
250
300
350
400
450
500
Accumulator Output
2
0
-2
0
50
100
150
200
250
300
350
400
450
500
Sample Number

Keyword-Only Parameters

Keyword-Only Parameters:

```
def add_these(x, y, *, k, m, z = 5):
    do_stuff()
```

As done above k, m and z are forced to be keyword only.

The interpreter knows to ignore the * position as part of the Python language syntax. (But *x would be a variable number of positional parameters followed by kw only)

Positional-Only Parameters

New in Python 3.8: Positional Only Keywords:

```
def add_these(x, y, /, k, m, z = 5):
    do_stuff()
```

As done above would force x and y to be positional only.

(This was previously reserved for certain built-in functions (as also indicated by a / in the parameter help for those functions)

Text Serialization (JSON)

JSON is a widely interoperable text stream serialization format that uses the json module from the standard library. JSON can only represent a subset of the Python built-in types.

Example:

```
import json

# to save
with open('my_vars.json','w') as f:
    json.dump([var1,var2], f)                # list of objs or single

# to restore
with open('my_vars.json', 'r' as f:
    var1, var2 = json.load(f)
```

Binary Serialization (Pickle)

Pickle is a python specific byte stream serialization format that uses the pickle module from the standard library. 'Pickling' serializes data to a byte-stream, and the data is recovered by 'unpickling', and can be used to save objects containing data to a file.

Example:

```
import pickle

# to save
with open('my_vars.pkl','wb') as f:
    pickle.dump([var1,var2], f)          # list of objs or single obj
# to restore
with open('my_vars.pkl', 'rb' as f:
    var1, var2 = pickle.load(f)
```

More Resources

When / Where to use Lambda Functions
https://treyhunner.com/2018/09/stop-writing-lambda-expressions/
(Great summary and helpful to avoid misuse)

Secrets of coding poorly:
https://docs.quantifiedcode.com/python-anti-patterns/index.html

Pipelining with Coroutines