MAX PLANCK INSTITUTE FOR **BIOLOGY OF AGEING**

Python workshop

bioinformatics@age.mpg.de

Why Python?

- Easy to learn.
- It's free and well-documented.
- Popular (easy to get help on internet forums), big community.
- Fast !?
- Scripts are portable (can run on windows, mac os or linux).
- A lot of libraries for many applications ready to use.

Basic syntax

- Python is a “high level” script language, which means it is as closest as possible to natural human language.
- It is possible to run commands interactively or in scripts.
- On the terminal, type python:

```
$ python
Python 2.7.12 (default, Feb  8 2017, 00:26:15)
[GCC 6.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Basic syntax

```
>>> print "Hello, Python!”
>>> print("Hello, Python!")
```

- Print is a function, usually in python all the functions are called by name_function()
- But before python 3, print function can be called without parenthesis ()
- Inside the parenthesis, for some functions, there are parameters, called arguments

Basic syntax

- Inside a script:

```
#!/usr/bin/python

print "Hello, Python!"
```

- First line points to the python interpreter on the operational system

- And now to run (on unix based system):

```
$ chmod +x test.py     # This is to make file executable
$./test.py
```

Basic syntax

- Using # on a script, you can write comments and they will be ignored for the execution of the program

- Blank lines are ignored in a script

- It is possible to write multiple statements on a single line, separated by ;

```
#!/usr/bin/python

print "Hello, Python!”
# This is just a comment
print("Hello World") ; x="abc";print(x)
```

- From now, in this workshop, we will use only iterative python or Jupyter notebook.

Variables

- A variable is something which can change!!
- It is a way of referring to a memory location by a computer program.
- It stores values, has a name (identifier) and data type (string, numbers, etc).
- While the program is running, the variable can be accessed, and sometimes can be changed.
- Python is not “strongly-typed” language. It means that the type of data storage on variables can be changed (other languages like C the variable should be declared with a data type).

Variables and identifiers

- Some people mistakes variables and identifiers.
- But identifiers are **names** of variables, they have name AND other features, like a value and data type.
- In addition, identifiers are not used only for variables, but also for functions, packages, etc.

In Python, a valid identifier is a non-empty sequence of characters of any length with:

- The start character can be the underscore "_" or a capital or lower case letter.
- The letters following the start character can be anything which is permitted as a start character plus the digits.
- Just a warning for Windows-spoilt users: Identifiers are case-sensitive!

- **Python keywords are not allowed as identifier names!** Ex: and, as, assert, break, class, continue, def, del, elif, else, except, for, if, in, is, lambda, not, or, pass, return, try, with.

Some examples of variables

```
i = 30

J = 32.1

some_string= "string"
```

Basic Data Types summary in Python

- Numerics:

1. int: integers, ex: 610, 9580
2. long: long integers of non-limited length (only python 2.x) ← Python 3 int is unlimited
3. Floating-point, ex: 42.11, 2.5415e-12
4. Complex, ex: x = 2 + 4i

- Sequences:

1. str: Strings (sequence of characters), ex: “ABCD”, “Hello world!”, “C_x-aer”
2. list
3. tuple

- Boolean: True or False
- Mapping – dict (dictionary)

Numbers

- int (integers): positive or negative numbers without decimal point.

```
a_number = 35
print type(a_number)
```

Note that there is no “ ”. If you include quotation marks:

```
a_number = "35"
```

It is not a integer (int) but a string (str) type.

- long (long integers): unlimited size integer to store very big numbers with a L in the and.

  Ex: 2025202222711317113121114112872826528289189181810500010120 21L

  This is only for python 2.x. Python 3 int is unlimited

Numbers

- float (floating point real values): real numbers with decimal points (ex: 23.567) and also can be in scientific notation (1.54e2 – it is the same of 2.5 x 10^2 and the same of 250)

```
a_float_number = 35.23

another_float_number=1.54e2
```

- complex (complex numbers): not used much in Python. They are of the form a + bJ (a and b are floats) and J an imaginary number

Number type conversions

- If x is a number:

```
x = 55.6
```

- convert to a int:

```
int(x)
```

- It is also possible to convert to any other type:

```
x = 55
float(x)
complex(x)
long(x)
```

- Also possible to convert to a string:

```
str(x)
```

Numerical operations

- Unlike for strings, for numeric data types the operators +,*,-,/ are arithmetic. Try it:

```
x = 55
y = 30
y + x
x - y
x * y
x / y
```

- **What if x = “55” and y = “30” ???**

- Some extra operators:

  % (Modulus) return the remainder of a division

  ** (Exponent) return result of exponential calculation

Strings

Strings are marked by quotes:

Wrapped with the single-quote (') character:

'This is a string with single quotes’

Wrapped with the double-quote (") character:

”Nick's dog is called Rex”

Wrapped with three characters, using either single-quote or double-quote:

'''A String in triple quotes can extend over multiple lines like this one, and can contain 'single' and "double" quotes.'''

Strings

A string in Python consists of a series or sequence of characters - letters, numbers, and special characters.

Strings can be indexed. The first character of a string has the index 0.

```
str_1 = "A string consists of characters"
str_1[0]
```

Output: ‘A’

```
str_1[3]
```

Output: ‘t’

Strings

And how to access the last character ?

The hard way:

- You need to discover the length of string
- You need to subtract 1 (because the index starts on zero)
- You need to use this value as index

```
str_1[len(str_1)-1]
```

But, Python is beautiful and we can do just:

```
str_1[-1]
```

Strings

- It is possible because the index can be also counted from the right, using negative values:

“STRING”

[-6],[-5],[-4],[-3],[-2],[-1]

In addition to the normal way, from the left:

“STRING”

[0],[1],[2],[3],[4],[5]

Strings are “immutable”

- Like in Java, Python strings cannot be changed.

```
s = "Some things are immutable!”
s[-1]=".”

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Operations with strings

- Concatenation: using operator + is possible to concatenate 2 or more strings:
  "Hello" + "World" will result in "HelloWorld" ← Attention to the absence of space

- Repetition: using operator * is possible to repeat n times one string:
  "HelloWorld" * 3 will result in "HelloWorldHelloWorldHelloWorld"

- Indexing: as mentioned before, is possible to recover one specific position of the string by index:
  "HelloWorld"[0] will result in "H"

Operations with strings

- Slicing: it is possible to recover substring of strings with slicing notation:

  “HelloWorld”[2:4] will result in “ll” ← Note that [2:4] will get the 2nd and 3rd position, not the 4th

- Size:

  len(“HelloWorld”) will result in 10 ← Note there is no quote marks “ ” because the result is a number

Operations with strings

- Split – It is possible to use a substring to split a string:

```
s = "Some things are immutable!”
print s.split()
print s.split(a)
s2 = "You can split some strings, and you can decide how to do it!”
print s2.split(",")
print s2.split(",")[0]
```

Escape sequences

- If you want to use some special characters with the literal meaning inside a string, you need to use backslash \ before.

  Ex: “The double quotation mark is \””

  If you use only ” without \ it will close the string without show ” inside the string

  Ex2: “I like to use \\”

  You need to use a backslash before a backslash to display \ inside a string

- Another scape sequences using backslash include:

  New line: \newline

  tabular space: \t ← very important dealing with tabular files

Lists

- It is a versatile data type in python and can be written as comma-separated values (items) limited by brackets [ ].

- Items in the list does not need to be of the **same type**!

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

- Similar to string, index of list also starts with 0 and the list can be sliced, concatenated, etc.

```
list1[-2]
list2[0:3]
```

Lists

- Unlike strings, it is possible to update a list by changing an element:

```
list1 = ['physics', 'chemistry', 1997, 2000]
print(list1[2])
list1[2] = 2001
print(list1[2])
```

- And also possible to delete a element of the list:

```
print(list1)
del list1[2]
print(list1)
```

Lists operations

- The same as string for + (concatenation) and * (repetition) and most of operators.
  Ex: len(list)

- More examples:

```
list1 = ['physics', 'chemistry', 1997, 2000, 2004, 1999,2000]
len(list1)
2001 in list1
2000 in list1
list1.count(2000)
list1.sort()
print(list1)
```

Tuple

- Tuple is very similar to list, but immutable.
- Instead [ ] it uses ( )

```
tup1 = ('bla','ble','blu',1,30)
tup2 = (1,2,3,4,5,6)
tup3=(50,)
```

Dictionary

- In the dictionary, for every item there is a key and a value, separated by colon (:)
- The items are separated by comma (,)
- Keys are unique in a dictionary, but the values may be not.
- You can access the values by the key

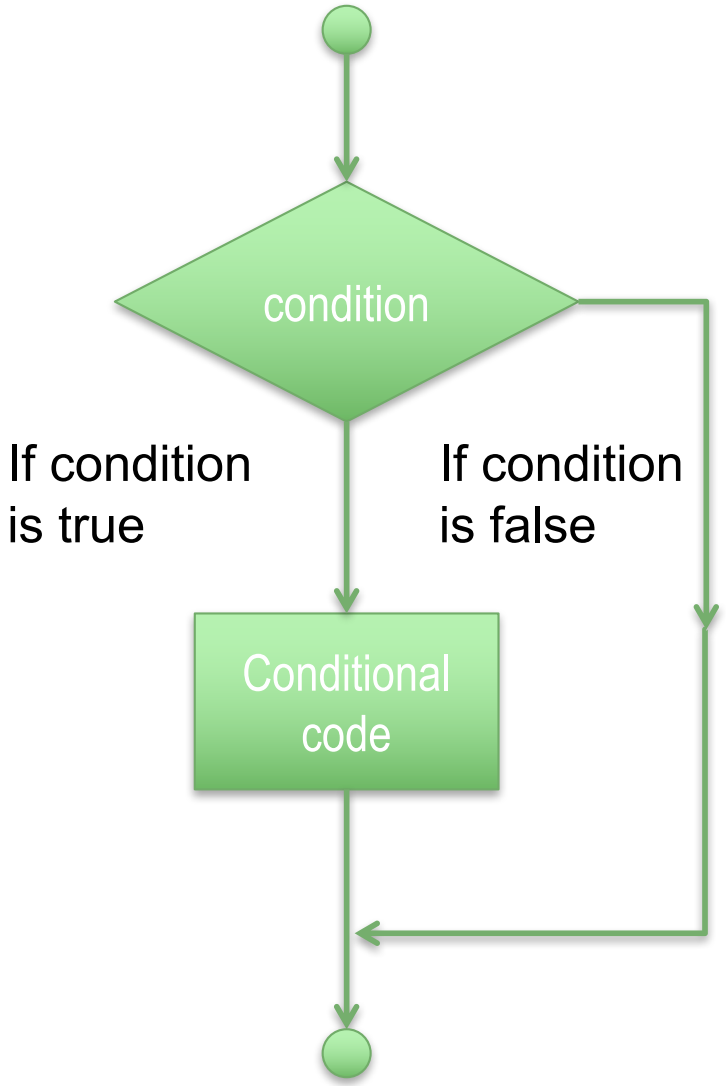
```
dict1 = {'Name': 'John', 'Age': 25, 'Class': 'First’}
dict1['Name']
dict1['Age']
```

Comparison and logical operators in Python

- **Logical:**
- and – both if both operators are true, the result is true
- or – on of the operators is true, the result is true
- not – reverse the logical operator

- **Comparison:**
- == check if two values are equal. Ex: a == b.
- != check if two values are NOT equal.
- > check if the left is greater than the right. Ex: a>b.
- < check if the left is less than right
- >= bigger or equal
- <= smaller or equal

Decision making



Decision making

- For decision making it is possible to use just if for one single condition, if and elif for more than one condition and else for anything else (and every elif).

```
var1=100
if var1>90:
        print "Variable  is greater than 90"
```

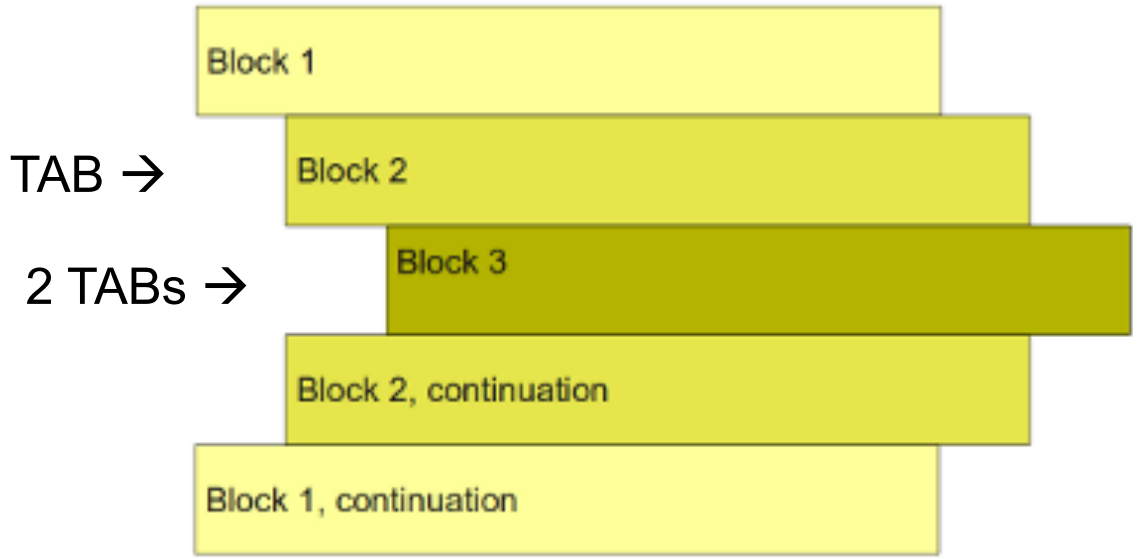
```
var1=100
if var1>90:
        print "Variable  is greater than 90"
else:
        print "Variable can be equal or smaller than 90"
```

```
var1=100
if var1>90:
        print "Variable is greater than 90"
elif var1<90:
        print "Variable is smaller than 90"
else:                                   #(could also be elif var1 == 90:)
        print "Variable is 90"
```

Indentation

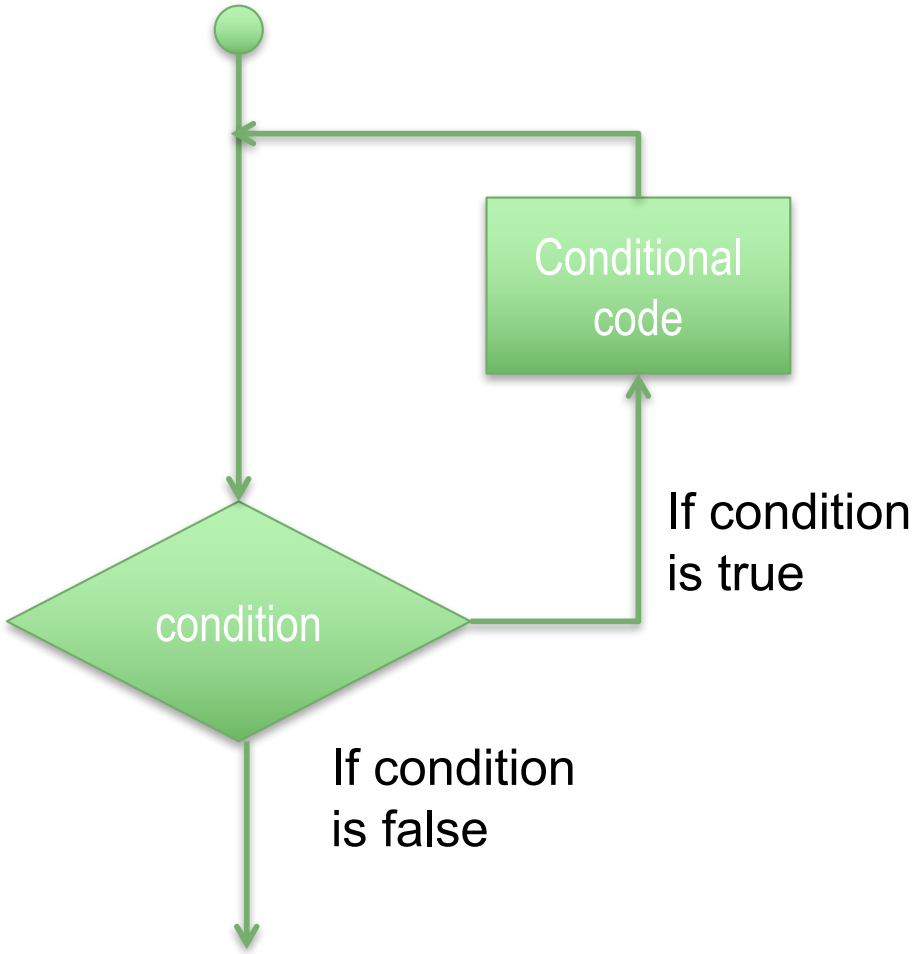
- **Attention for the indentation in Python:**

- No braces or reserved words to indicate blocks of code for class, functions or flow control. It is done by indentation.



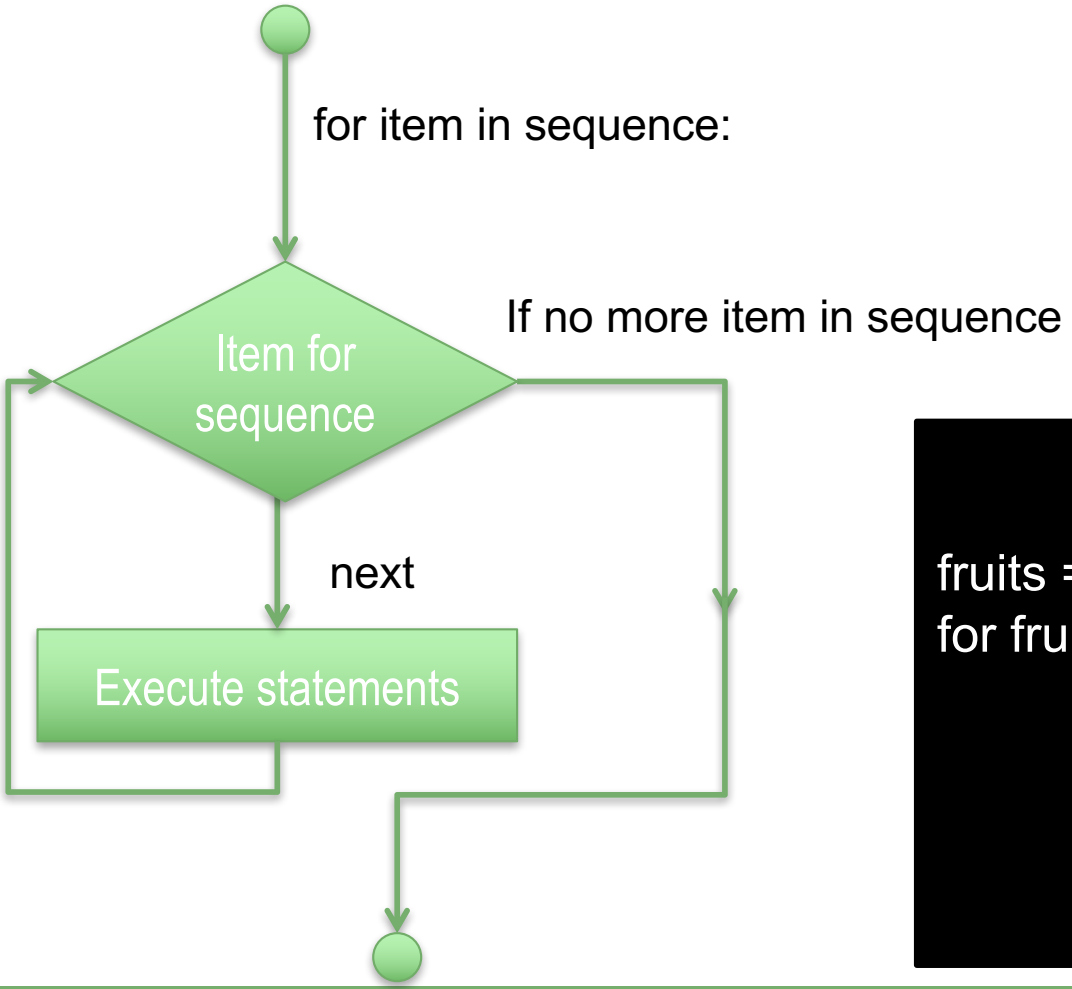
Loops

- Sometimes you need to repeat the same operations, so it is possible to use a loop, with a structure of repetition based in a decision or just a fixed number of times.



Loops

For loop has the ability to iterate over the items of any sequence, such as a list or a string.



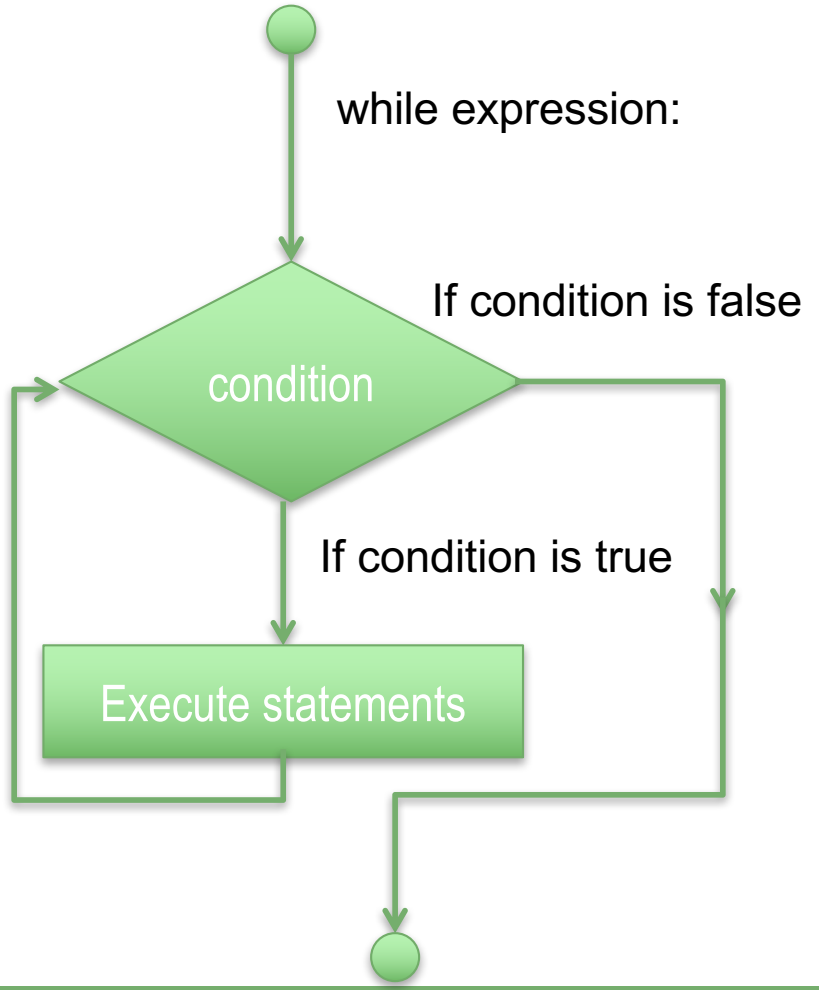

```
for letter in 'Python':
        print 'Current Letter :', letter


fruits = ["apple", "mango", "banana"]
for fruit in fruits:
        print  'Current fruit :', fruit
```

```
fruits = ["apple", "mango", "banana"]
for fruit in fruits:
        for letter in fruit:
                print 'Current fruit :', fruit,  \
                'Current letter :', letter
```

Loops

```
count = 0
while (count < 9):
        print  "The count is:", count
        count = count+1
print "Bye!"
```

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.



Loops

- The infinite loop: you need to take care when using while loops to not create an endless loop (unless you need one).

```
var =1
while var ==1:
            print  "Endless loop"
```

- To stop it you need to use CTRL+C

I/O in Python

- A program (or script) needs to deal with an input and to generate an output. In Python there are many ways to input data and we will see the most basic ones first.

```
str = raw_input("Enter your input: ");
print "Received input is : ", str
```

The raw_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline).

I/O in Python

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
str = input("Enter your input: ");
print "Received input is : ", str
```

Example of input:

[x*5 for x in range(2,10,2)]

Output:

[10, 20, 30, 40]

I/O in Phyton - files

- Before read or write a file, you have to open it with the open() function.
- This function create a file object.

```
afile = open("example.txt", "w+")
```

- The first argument is the file name, and the second is the “access mode”: It determines if the file will be only read, if can be write, append, etc. The default is r (read). In the example, w+ is for both writing and reading, overwriting the existing file if the file exists.
- A good practice is to close the file after do what you need to do, using close().

```
afile.close()
```

I/O in Phyton - files

- The file object has methods to make easier to work with it. For example read() and write().

```
# Read a file
fo = open("file1.txt", "r+")
str = fo.read();
print "Read String is : ", str
fo.close()
```

```
# Write to a file
fo = open("file1.txt", "wb")
fo.write( "Python is amazing.\nYeah its great!!\n")
fo.close()
```

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like sum(), print(), etc. but you can also create your own functions. These functions are called user-defined functions.

You can define functions to provide the required functionality.

Functions

Function blocks begin with the keyword *def* followed by the function name and parentheses (( )).

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

```
def functionname( parameters ):
```

Functions

- The code block within every function starts with a colon (:) and is indented.

- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

- The first statement of a function can be an optional statement - the documentation string of the function or docstring.

```
def printme( str ):
   # Optional documentation about the function
   "This prints a passed string into this function"
   print str
   return
```

Functions

- Calling the fuction:

```
# Now you can call printme function

printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Functions

- It is possible to use arguments as keyword arguments. It allows you to skip arguments or place out of order because python will identify by the keyword

```
# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

Functions

- It is possible to use default arguments.

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Functions

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
# Function definition is here
def sum( arg1, arg2 ):
   # Sum both parameters
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

Functions

- Global vs. Local variables:

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

```
total = 0; # Global variable.

# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.

A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

The Python code for a module named aname normally resides in a file named aname.py. Here's an example of a simple module, support.py

```
def print_func( name ):
   print "Hello : ", name
   return
```

Modules

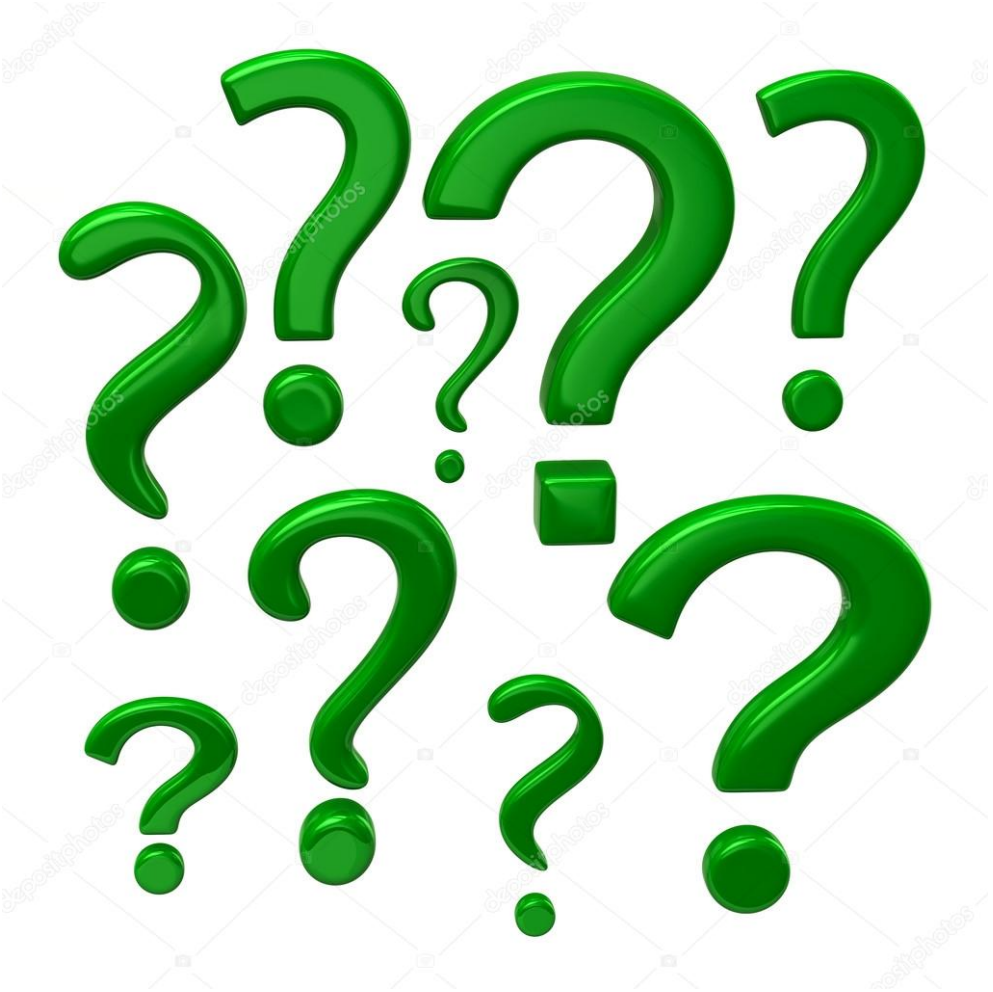
- To use a module you need first to import.

```
# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

- You can also use an alias

```
# Import module support with an alias “sup”
import support as sup
```

Questions ?



Second day

- Numerical operations (Numpy)
- Dataframes (pandas)
- Plots (including heatmap and PCA) – matplotlib and seaborn

Numerical operations with Numpy

- Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays (including many numerical operations)

- We will not cover arrays in detail, but we will see how to use numpy to do numerical operations

- NumPy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality.

```
import numpy as np
a = np.array( [ [1,2], [3,4] ] )
```

Numerical operations with Numpy

```
import numpy as np

a = np.arange(15).reshape(3,5)

a
```

output: array([[0, 1, 2, 3, 4],
[5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]])

```
a = np.array([2,3,4])
b = np.array([2.1,3.2,4.5])
c = np.array( [ [1,2], [3,4] ] )
```

Numerical operations with Numpy

```
c = a+b
d=a-b
d**2
a*b         #(elementwise product)
a.dot(b)  #(matrix product)
```

```
b=np.arange(12).reshape(3,4)
b.sum(axis=0)
b.min(axis=1)
b.cumsum(axis=1)
b.T
```

```
b[2,3]
b[-1]
```

Introduction to data frames with Pandas

What is a Pandas data frame?

It is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).

Arithmetic operations align on both row and column labels.

Out[34]:

	GEOID	2005	2006	2007	2008	2009	2010	2011	2012	2013
State										
Alabama	04000US01	37150	37952	42212	44476	39980	40933	42590	43464	41381
Alaska	04000US02	55891	56418	62993	63989	61604	57848	57431	63648	61137
Arizona	04000US04	45245	46657	47215	46914	45739	46896	48621	47044	50602
Arkansas	04000US05	36658	37057	40795	39586	36538	38587	41302	39018	39919
California	04000US06	51755	55319	55734	57014	56134	54283	53367	57020	57528

Source: http://pythonhow.com/

Introduction to data frames with Pandas

- In general, we import a tabular or CSV file to a pandas dataframe

```
import pandas as pd
df = pd.read_csv("file.csv")
df2 = pd.read_table("file.tabular")
df3= pd.read_excel("file.xls")
```

- The are some options, for example, we can use different separator character, we can read a file with or without a header

```
import pandas as pd
df = pd.read_csv("file.csv", sep="\s", header=None)
```

Files provided

- I provided 2 files:
- old_avs.csv and metatable.csv

- Metagenomics analysis: AVS is Average Genome Size calculated based on gene markers for each metagenomic sample

- Metatable is metadata about each sample

Open the files

- df=pd.read_table("old_avs.csv",header=None)

Introduction to data frames with Pandas

```
df.head()
```

Out[11]:

	4494598	7897771.17488	1.4879709401
0	4494599	6.603478e+06	9.012739
1	4494600	6.130061e+06	7.906269
2	4494601	6.231407e+06	14.234019
3	4494602	5.530277e+06	1.558494
4	4494603	1.271626e+07	5.553935

Header?

Out[28]:

	0	1	2
0	4494598	7.897771e+06	1.487971
1	4494599	6.603478e+06	9.012739
2	4494600	6.130061e+06	7.906269
3	4494601	6.231407e+06	14.234019
4	4494602	5.530277e+06	1.558494

Introduction to data frames with Pandas

```
df.columns=["Sample","RPKG","Genome Equivalents"]
df.head()
```

Out[31]:

	Sample	RPKG	Genome Equivalents
0	4494598	7.897771e+06	1.487971
1	4494599	6.603478e+06	9.012739
2	4494600	6.130061e+06	7.906269
3	4494601	6.231407e+06	14.234019
4	4494602	5.530277e+06	1.558494

Header!!

Introduction to data frames with Pandas

```
#only display new dataframe with new index
df.set_index("Sample")
#to modify the original dataframe:
df.set_index("Sample", inplace=True)
```

Index

Out[34]:

Sample	RPKG	Genome Equivalents
4494598	7.897771e+06	1.487971
4494599	6.603478e+06	9.012739
4494600	6.130061e+06	7.906269
4494601	6.231407e+06	14.234019
4494602	5.530277e+06	1.558494
4494603	1.271626e+07	5.553935
4494604	1.607910e+07	8.815452

Introduction to data frames with Pandas

```
df.describe()
```

Out[32]:

	RPKG	Genome Equivalents
count	7.600000e+01	76.000000
mean	6.387378e+06	95.323624
std	4.437504e+06	346.565560
min	1.699219e+06	1.487971
25%	3.291981e+06	12.358437
50%	4.863723e+06	26.123614
75%	7.972981e+06	52.186570
max	2.237441e+07	2910.334469

Introduction to data frames with Pandas

```
df.dtypes
```

```
Out[36]: Sample                    object
         RPKG                     float64
         Genome Equivalents       float64
         dtype: object
```

```
df.astype(str)
```

Introduction to data frames with Pandas

```
df.T
```

Out[39]:

	0	1	2	3	4	5	6	7	8	9	...	
Sample	4494598	4494599	4494600	4494601	4494602	4494603	4494604	4494605	4494606	4494607	...	CAM_SMPL
RPKG	7.89777e+06	6.60348e+06	6.13006e+06	6.23141e+06	5.53028e+06	1.27163e+07	1.60791e+07	1.56391e+07	1.37189e+07	4.77491e+06	...	8.30
Genome Equivalents	1.48797	9.01274	7.90627	14.234	1.55849	5.55393	8.81545	6.93693	9.31441	9.67757	...	

3 rows × 76 columns

Introduction to data frames with Pandas

```
df.sort_values(by="Genome Equivalents")
#df.sort_values(by='Genome Equivalents',ascending=False)
```

Out[41]:

	Sample	RPKG	Genome Equivalents
0	4494598	7.897771e+06	1.487971
4	4494602	5.530277e+06	1.558494
28	CAM_SMPL_003358	1.591921e+07	4.732174
38	CAM_SMPL_003373	1.770188e+07	4.967061
5	4494603	1.271626e+07	5.553935
7	4494605	1.563906e+07	6.936927
2	4494600	6.130061e+06	7.906269
36	CAM_SMPL_003370	2.237441e+07	8.597071
6	4494604	1.607910e+07	8.815452
1	4494599	6.603478e+06	9.012739
8	4494606	1.371887e+07	9.314405

Introduction to data frames with Pandas

```
df["RPKG"]
```

```
df[["RPKG","Genome Equivalents"]]
```

```
df.iloc[0:3]
```

Out[46]:

	Sample	RPKG	Genome Equivalents
0	4494598	7.897771e+06	1.487971
1	4494599	6.603478e+06	9.012739
2	4494600	6.130061e+06	7.906269

Introduction to data frames with Pandas

```
df[df["Genome Equivalents"]>100]
```

Out[51]:

	Sample	RPKG	Genome Equivalents
12	4539290_FL	2.450922e+06	143.015812
14	4539502	1.783111e+06	842.126227
16	4539504	2.219105e+06	384.278247
19	4539507	1.825381e+06	137.715882
20	4539508	3.281944e+06	117.906836
53	CAM_SMPL_003393	1.813627e+06	118.742288
73	FL_clean	5.766925e+06	355.130579
74	IMG_db	8.184903e+06	2910.334469
75	PA_clean	7.796310e+06	304.891751

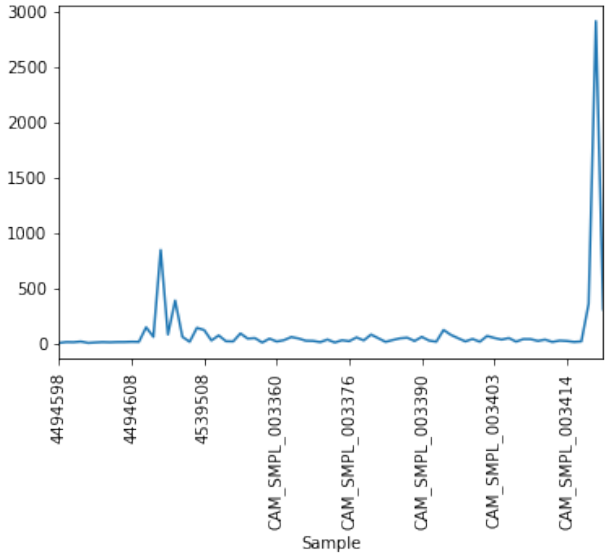
Introduction to data frames with Pandas

```
df[“Genome Equivalents”].min(axis=1)
df[“Genome Equivalents”].max(axis=0)
df[“Genome Equivalents”].max()-df[“Genome Equivalents”].min()

df[“Genome Equivalents”]+1
df[“new col”]=df[“Genome Equivalents”]+5

df[“new col 2”]=(df[“Genome Equivalents”]+df[“RPKG”])/10
```

Plots with matplotlib

The easy (and ugly) way:

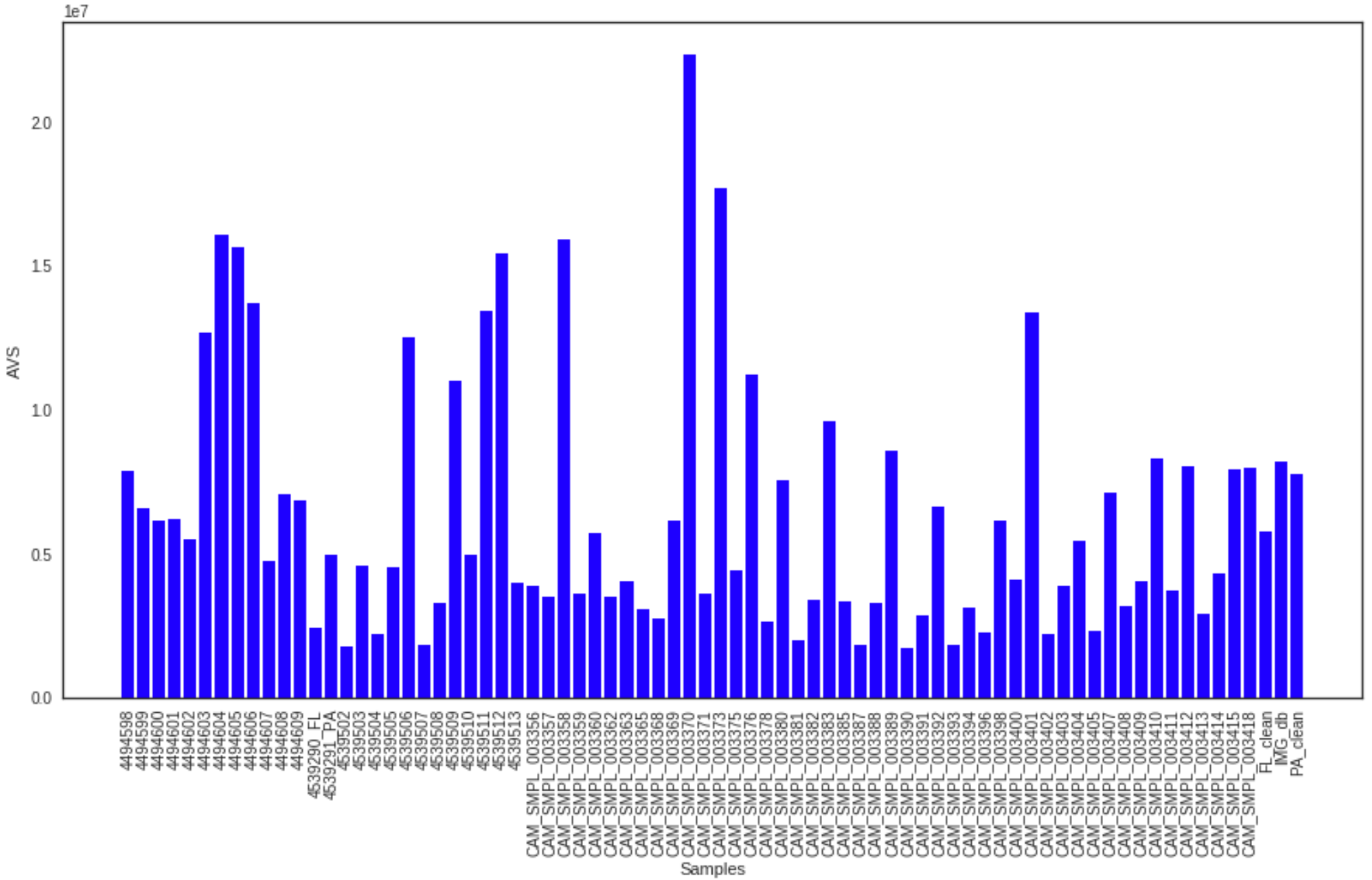
```
Import matplotlib.pyplot as plt

df_tmp=df.set_index("Sample")
df_tmp["Genome Equivalents"].plot()
plt.xticks(rotation=90)
plt.show()
```



Plots with matplotlib

Some nicer way:

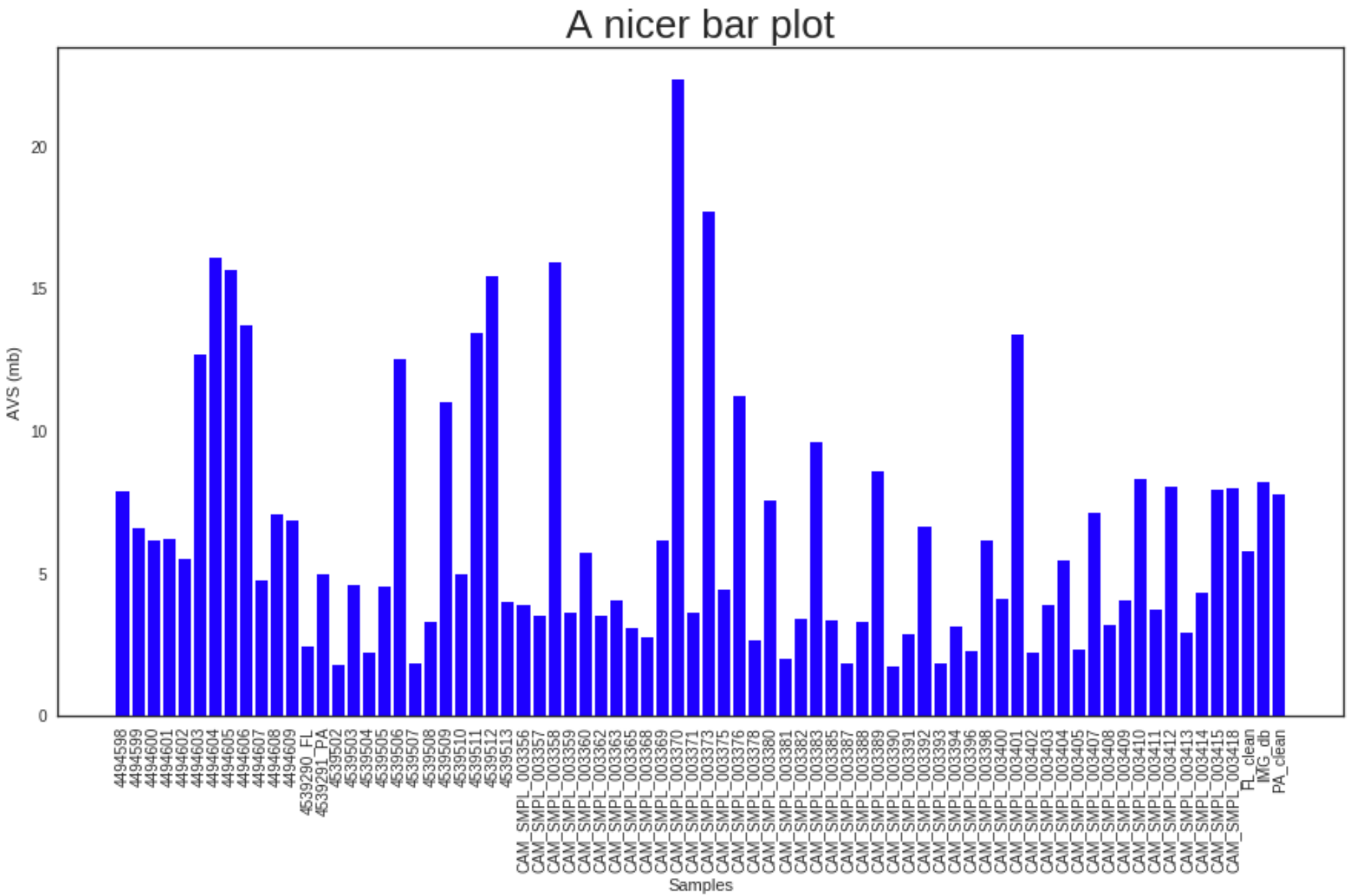
```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("white")
plt.figure(figsize=(15,8))

x=range(len(df["AVS"]))
y=df["AVS"]
labels=df["Sample"]

plt.bar(x,y,color="b")
plt.ylabel("AVS")
plt.xlabel("Samples")
plt.xticks(x, labels ,rotation="vertical")
plt.show()
```


1e7
2.0
1.5
1.0
0.5
0.0
AVS
4494598
4494599
4494600
4494601
4494602
4494603
4494604
4494605
4494606
4494607
4494608
4494609
4539290_FL
4539291_PA
4539502
4539503
4539504
4539505
4539506
4539507
4539508
4539509
4539510
4539511
4539512
4539513
CAM_SMPL_003356
CAM_SMPL_003357
CAM_SMPL_003358
CAM_SMPL_003359
CAM_SMPL_003360
CAM_SMPL_003362
CAM_SMPL_003363
CAM_SMPL_003365
CAM_SMPL_003368
CAM_SMPL_003369
CAM_SMPL_003370
CAM_SMPL_003371
CAM_SMPL_003373
CAM_SMPL_003375
CAM_SMPL_003376
CAM_SMPL_003378
CAM_SMPL_003380
CAM_SMPL_003381
CAM_SMPL_003382
CAM_SMPL_003383
CAM_SMPL_003385
CAM_SMPL_003387
CAM_SMPL_003388
CAM_SMPL_003389
CAM_SMPL_003390
CAM_SMPL_003391
CAM_SMPL_003392
CAM_SMPL_003393
CAM_SMPL_003394
CAM_SMPL_003396
CAM_SMPL_003398
CAM_SMPL_003400
CAM_SMPL_003401
CAM_SMPL_003402
CAM_SMPL_003403
CAM_SMPL_003404
CAM_SMPL_003405
CAM_SMPL_003407
CAM_SMPL_003408
CAM_SMPL_003409
CAM_SMPL_003410
CAM_SMPL_003411
CAM_SMPL_003412
CAM_SMPL_003413
CAM_SMPL_003414
CAM_SMPL_003415
CAM_SMPL_003418
FL_clean
IMG_db
PA_clean
Samples

Plots with matplotlib

Some even nicer way:

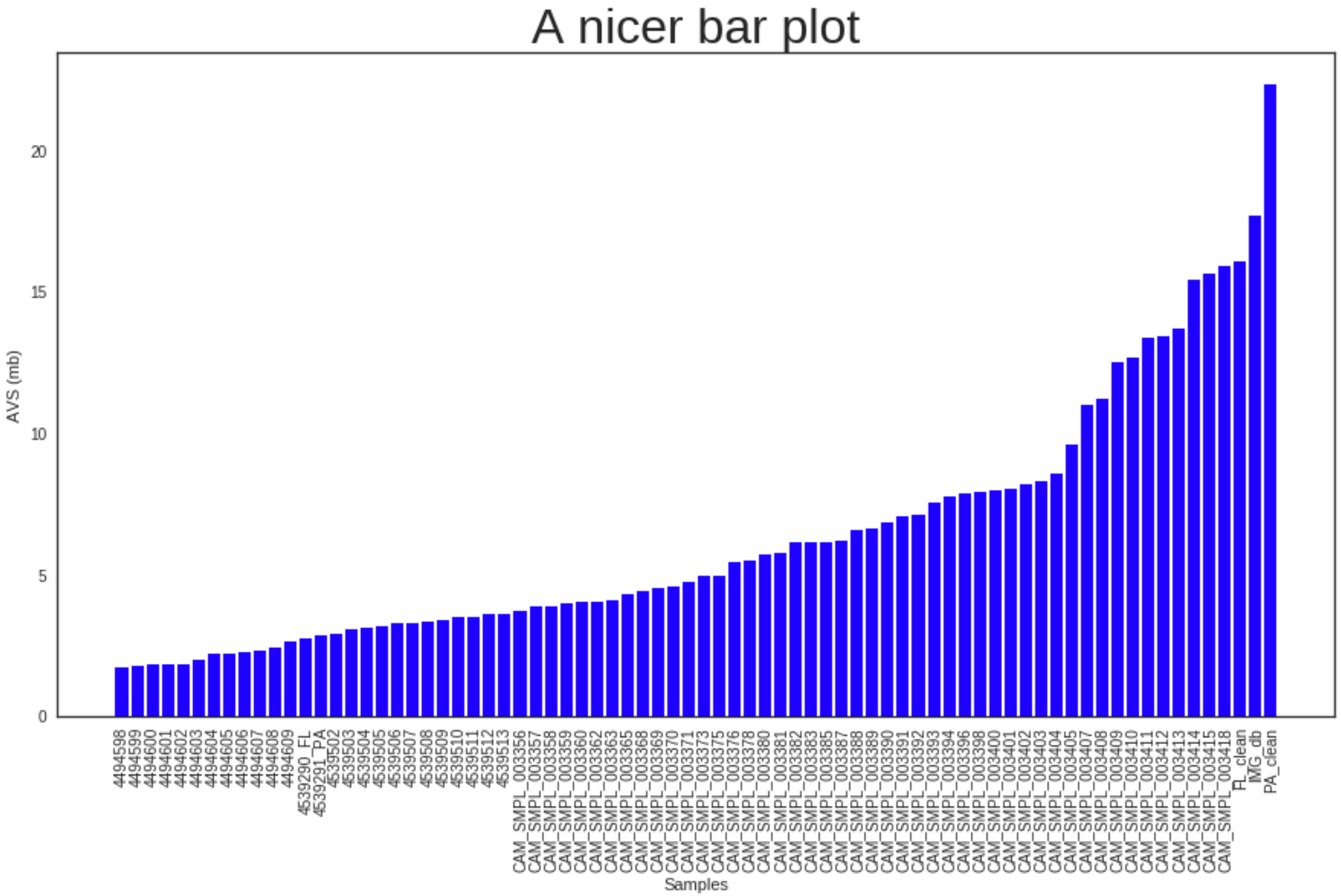
```
df["AVS (mb)"]=df["AVS"]/1000000

sns.set_style("white")
plt.figure(figsize=(15,8))

x=range(len(df["AVS (mb)"]))
y=df["AVS (mb)"]
labels=df["Sample"]

plt.bar(x,y,color="b")
plt.ylabel("AVS (mb)")
plt.xlabel("Samples")
plt.xticks(x, labels ,rotation="vertical")
plt.title("A nicer bar plot",fontsize=32)
plt.show()
```


A nicer bar plot
AVS (mb)
0
5
10
15
20
4494598
4494599
4494600
4494601
4494602
4494603
4494604
4494605
4494606
4494607
4494608
4494609
4539290_FL
4539291_PA
4539502
4539503
4539504
4539505
4539506
4539507
4539508
4539509
4539510
4539511
4539512
4539513
CAM_SMPL_003356
CAM_SMPL_003357
CAM_SMPL_003358
CAM_SMPL_003359
CAM_SMPL_003360
CAM_SMPL_003362
CAM_SMPL_003363
CAM_SMPL_003365
CAM_SMPL_003368
CAM_SMPL_003369
CAM_SMPL_003370
CAM_SMPL_003371
CAM_SMPL_003373
CAM_SMPL_003375
CAM_SMPL_003376
CAM_SMPL_003378
CAM_SMPL_003380
CAM_SMPL_003381
CAM_SMPL_003382
CAM_SMPL_003383
CAM_SMPL_003385
CAM_SMPL_003387
CAM_SMPL_003388
CAM_SMPL_003389
CAM_SMPL_003390
CAM_SMPL_003391
CAM_SMPL_003392
CAM_SMPL_003393
CAM_SMPL_003394
CAM_SMPL_003396
CAM_SMPL_003398
CAM_SMPL_003400
CAM_SMPL_003401
CAM_SMPL_003402
CAM_SMPL_003403
CAM_SMPL_003404
CAM_SMPL_003405
CAM_SMPL_003407
CAM_SMPL_003408
CAM_SMPL_003409
CAM_SMPL_003410
CAM_SMPL_003411
CAM_SMPL_003412
CAM_SMPL_003413
CAM_SMPL_003414
CAM_SMPL_003415
CAM_SMPL_003418
FL_clean
IMG_db
PA_clean
Samples

Plots with matplotlib

Some even nicer way:

```
sns.set_style("white")
plt.figure(figsize=(15,8))

x=range(len(df.sort_values("AVS")["AVS (mb)"]))
y=df.sort_values("AVS")["AVS (mb)"]
label=df["Sample"]

plt.bar(x,y,color="b")
plt.ylabel("AVS (mb)")
plt.xlabel("Samples")
plt.xticks(x, label,rotation="vertical")
plt.title("A nicer bar plot",fontsize=32)

plt.show()
```

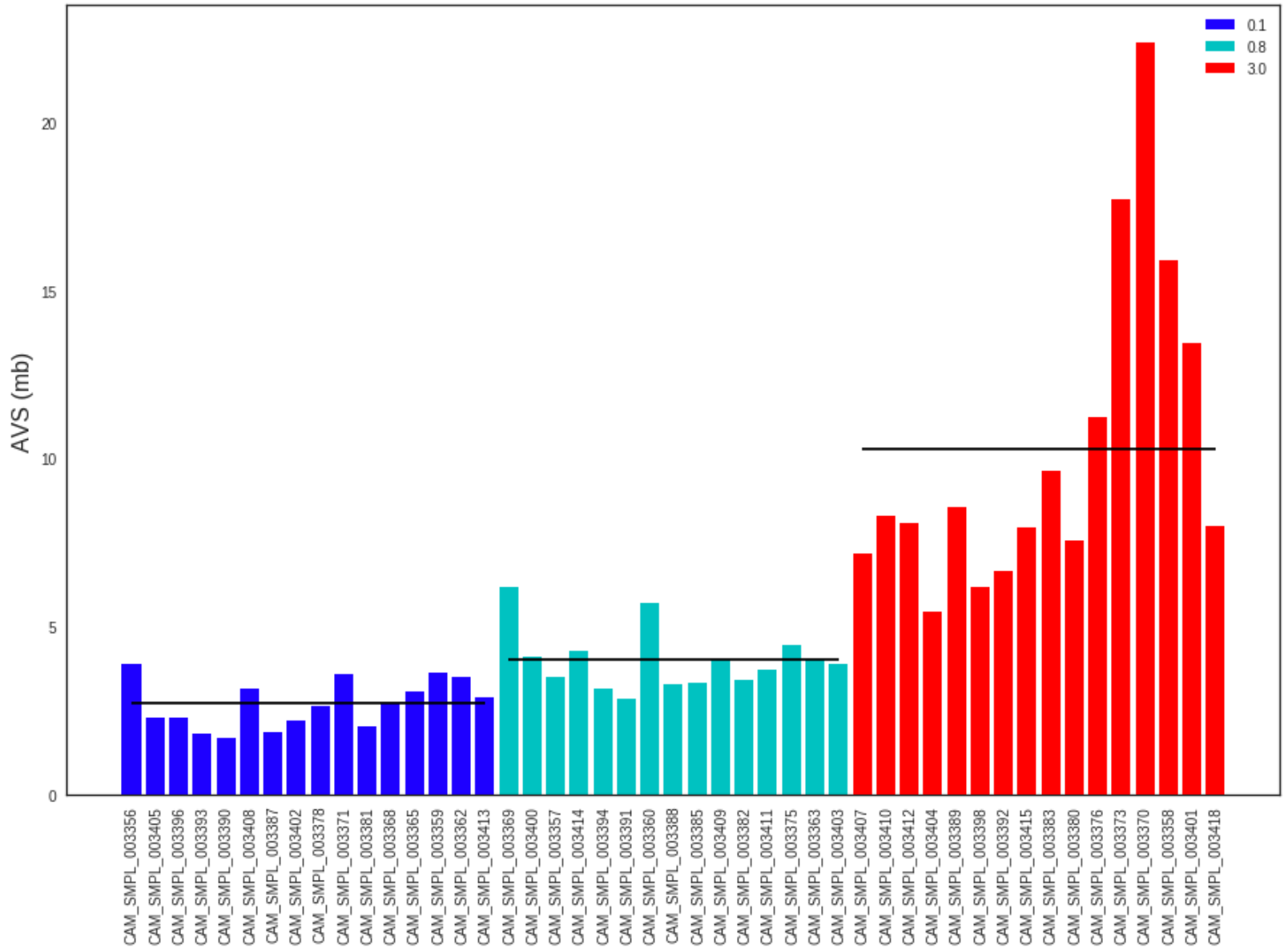

A nicer bar plot
AVS (mb)
0
5
10
15
20
4494598
4494599
4494600
4494601
4494602
4494603
4494604
4494605
4494606
4494607
4494608
4494609
4539290_FL
4539291_PA
4539502
4539503
4539504
4539505
4539506
4539507
4539508
4539509
4539510
4539511
4539512
4539513
CAM_SMPL_003356
CAM_SMPL_003357
CAM_SMPL_003358
CAM_SMPL_003359
CAM_SMPL_003360
CAM_SMPL_003362
CAM_SMPL_003363
CAM_SMPL_003365
CAM_SMPL_003368
CAM_SMPL_003369
CAM_SMPL_003370
CAM_SMPL_003371
CAM_SMPL_003373
CAM_SMPL_003375
CAM_SMPL_003376
CAM_SMPL_003378
CAM_SMPL_003380
CAM_SMPL_003381
CAM_SMPL_003382
CAM_SMPL_003383
CAM_SMPL_003385
CAM_SMPL_003387
CAM_SMPL_003388
CAM_SMPL_003389
CAM_SMPL_003390
CAM_SMPL_003391
CAM_SMPL_003392
CAM_SMPL_003393
CAM_SMPL_003394
CAM_SMPL_003396
CAM_SMPL_003398
CAM_SMPL_003400
CAM_SMPL_003401
CAM_SMPL_003402
CAM_SMPL_003403
CAM_SMPL_003404
CAM_SMPL_003405
CAM_SMPL_003407
CAM_SMPL_003408
CAM_SMPL_003409
CAM_SMPL_003410
CAM_SMPL_003411
CAM_SMPL_003412
CAM_SMPL_003413
CAM_SMPL_003414
CAM_SMPL_003415
CAM_SMPL_003418
FL_clean
IMG_db
PA_clean
Samples

Merging dataframes on Pandas

```
df2=pd.read_csv("../metatable.csv")

merged=pd.merge(df,df2,left_on="Sample",right_on="SAMPLE_ACC")
merged.dropna(how="all",axis=1)

pd.merge(df,df2,left_on="Sample",right_on="SAMPLE_ACC",how="outer")

merged=merged.sort_values("FILTER_MIN")
merged.reset_index(inplace=True)


g1_ = merged[merged["FILTER_MIN"]==0.1]
g2_ = merged[merged["FILTER_MIN"]==0.8]
g3_ = merged[merged["FILTER_MIN"]==3.0]
```

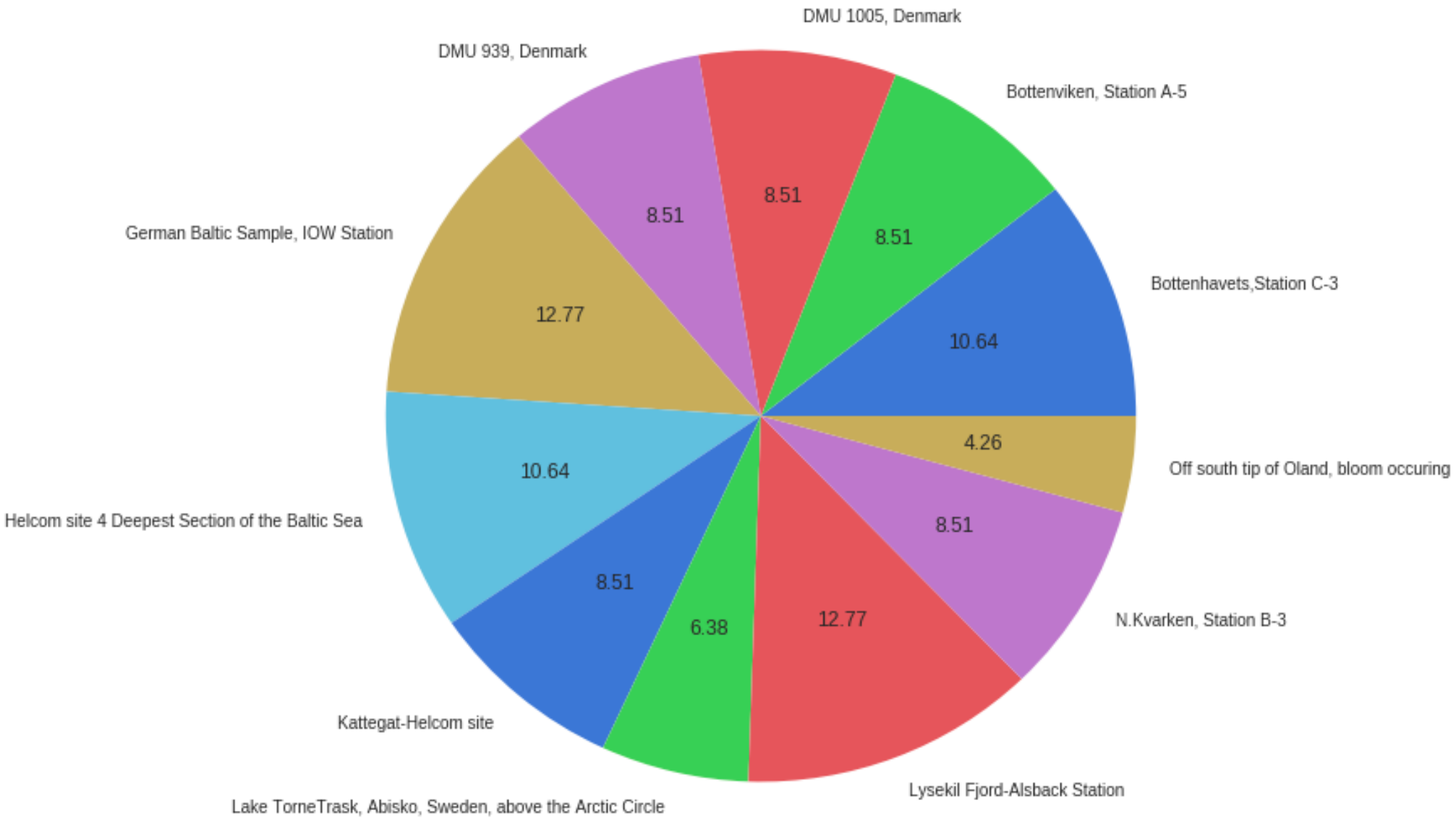
Plots with matplotlib

```
sns.set_style("white")
plt.figure(figsize=(15,10))
x1=g1_.index
y1=g1_["AVS (mb)"]
x2=g2_.index
y2=g2_["AVS (mb)"]
x3=g3_.index
y3=g3_["AVS (mb)"]
plt.bar(x1,y1,color="b")
plt.bar(x2,y2,color="c")
plt.bar(x3,y3,color="r")
plt.ylabel("AVS (mb)",size="16")
plt.xticks(range(len(merged)), merged["Sample"], size='small',rotation="vertical")
plt.legend(merged["FILTER_MIN"].drop_duplicates())
plt.hlines(y1.astype(float).mean(),x1[0],x1[-1])
plt.hlines(y2.astype(float).mean(),x2[0],x2[-1])
plt.hlines(y3.astype(float).mean(),x3[0],x3[-1])
plt.show()
```


0.1
0.8
3.0
AVS (mb)
0
5
10
15
20
CAM_SMPL_003356
CAM_SMPL_003405
CAM_SMPL_003396
CAM_SMPL_003393
CAM_SMPL_003390
CAM_SMPL_003408
CAM_SMPL_003387
CAM_SMPL_003402
CAM_SMPL_003378
CAM_SMPL_003371
CAM_SMPL_003381
CAM_SMPL_003368
CAM_SMPL_003365
CAM_SMPL_003359
CAM_SMPL_003362
CAM_SMPL_003413
CAM_SMPL_003369
CAM_SMPL_003400
CAM_SMPL_003357
CAM_SMPL_003414
CAM_SMPL_003394
CAM_SMPL_003391
CAM_SMPL_003360
CAM_SMPL_003388
CAM_SMPL_003385
CAM_SMPL_003409
CAM_SMPL_003382
CAM_SMPL_003411
CAM_SMPL_003375
CAM_SMPL_003363
CAM_SMPL_003403
CAM_SMPL_003407
CAM_SMPL_003410
CAM_SMPL_003412
CAM_SMPL_003404
CAM_SMPL_003389
CAM_SMPL_003398
CAM_SMPL_003392
CAM_SMPL_003415
CAM_SMPL_003383
CAM_SMPL_003380
CAM_SMPL_003376
CAM_SMPL_003373
CAM_SMPL_003370
CAM_SMPL_003358
CAM_SMPL_003401
CAM_SMPL_003418

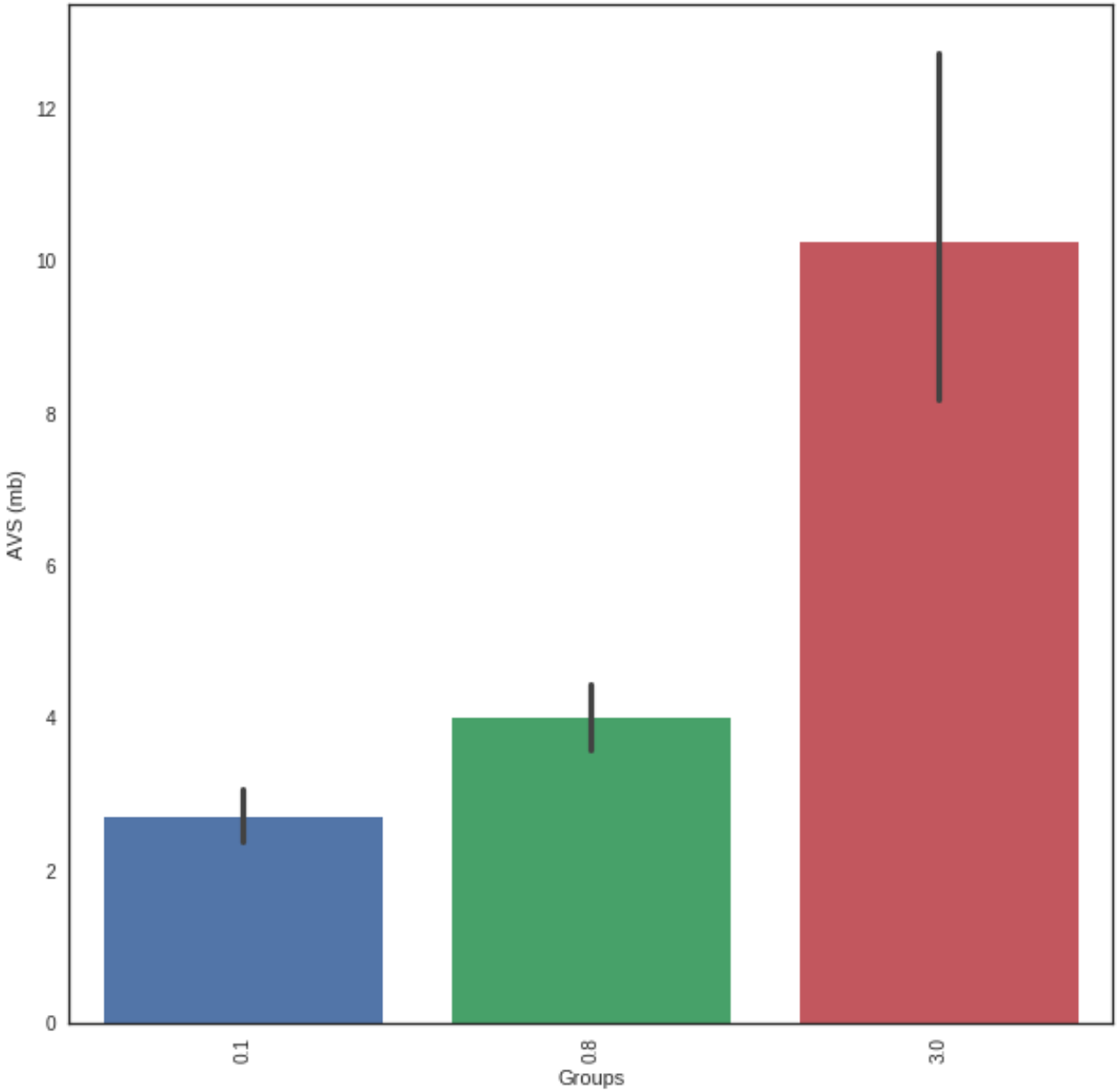
Plots with matplotlib

```
sns.set_style("white")
plt.figure(figsize=(10,10))

values=merged.groupby("SITE_NAME")["Sample"].count()
lab=values.index

plt.pie(values,labels=lab,autopct='%1.2f')
plt.show()
```


DMU 1005, Denmark
DMU 939, Denmark
Bottenviken, Station A-5
German Baltic Sample, IOW Station
Bottenhavets,Station C-3
Off south tip of Oland, bloom occuring
Helcom site 4 Deepest Section of the Baltic Sea
N.Kvarken, Station B-3
Kattegat-Helcom site
Lysekil Fjord-Alsback Station
Lake TorneTrask, Abisko, Sweden, above the Arctic Circle
8.51
8.51
8.51
12.77
10.64
4.26
10.64
8.51
8.51
6.38
12.77

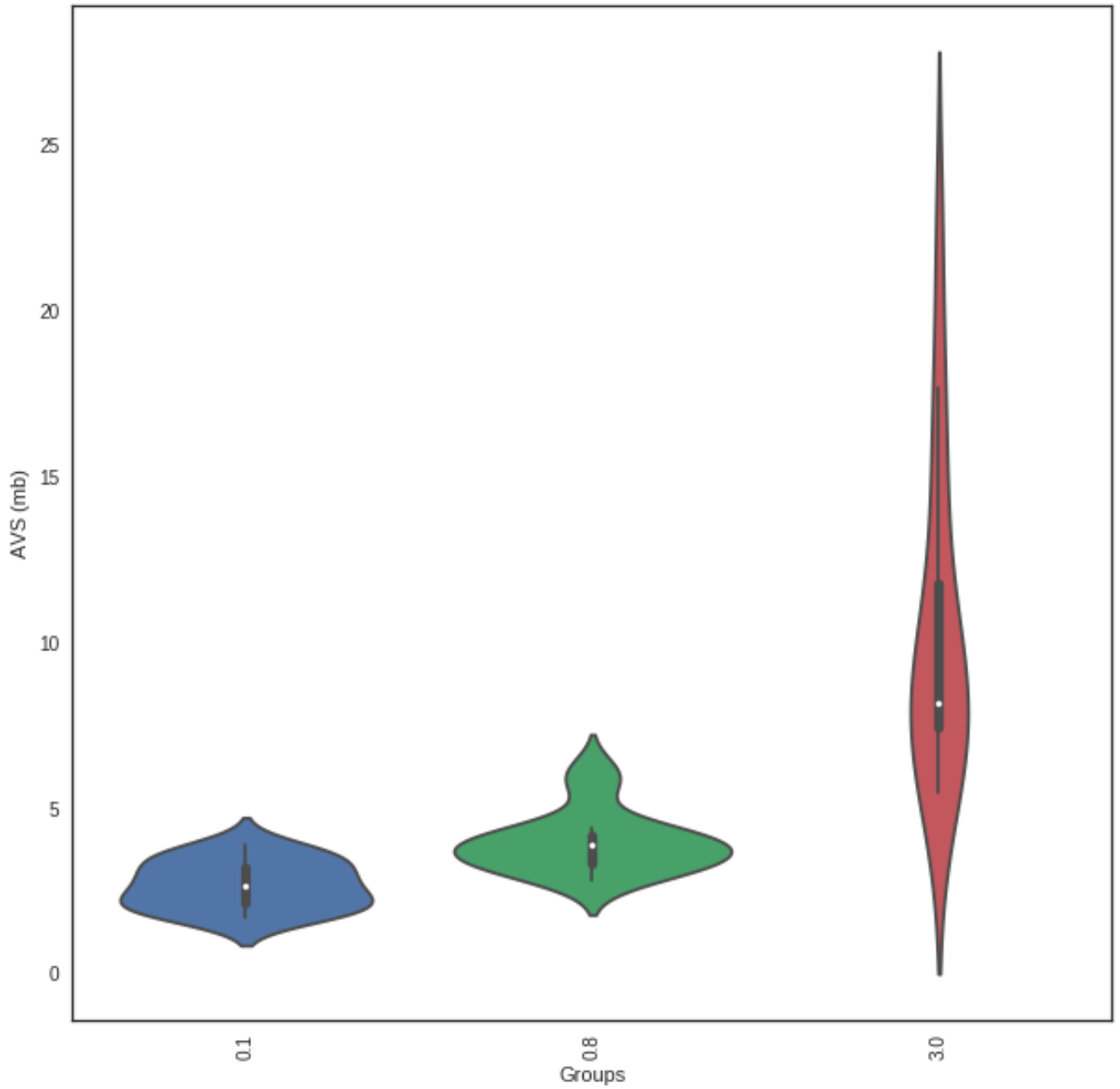
Plots with seaborn

```
sns.set_style("white")
plt.figure(figsize=(10,10))

x=merged["FILTER_MIN"]
y=merged["AVS (mb)"]

sns.barplot(x,y)
plt.ylabel("AVS (mb)")
plt.xlabel("Groups")
plt.xticks(range(len(x.drop_duplicates())), x.drop_duplicates(),rotation="vertical")
plt.show()
```


12
10
8
6
4
2
0
AVS (mb)
0.1
0.8
3.0
Groups

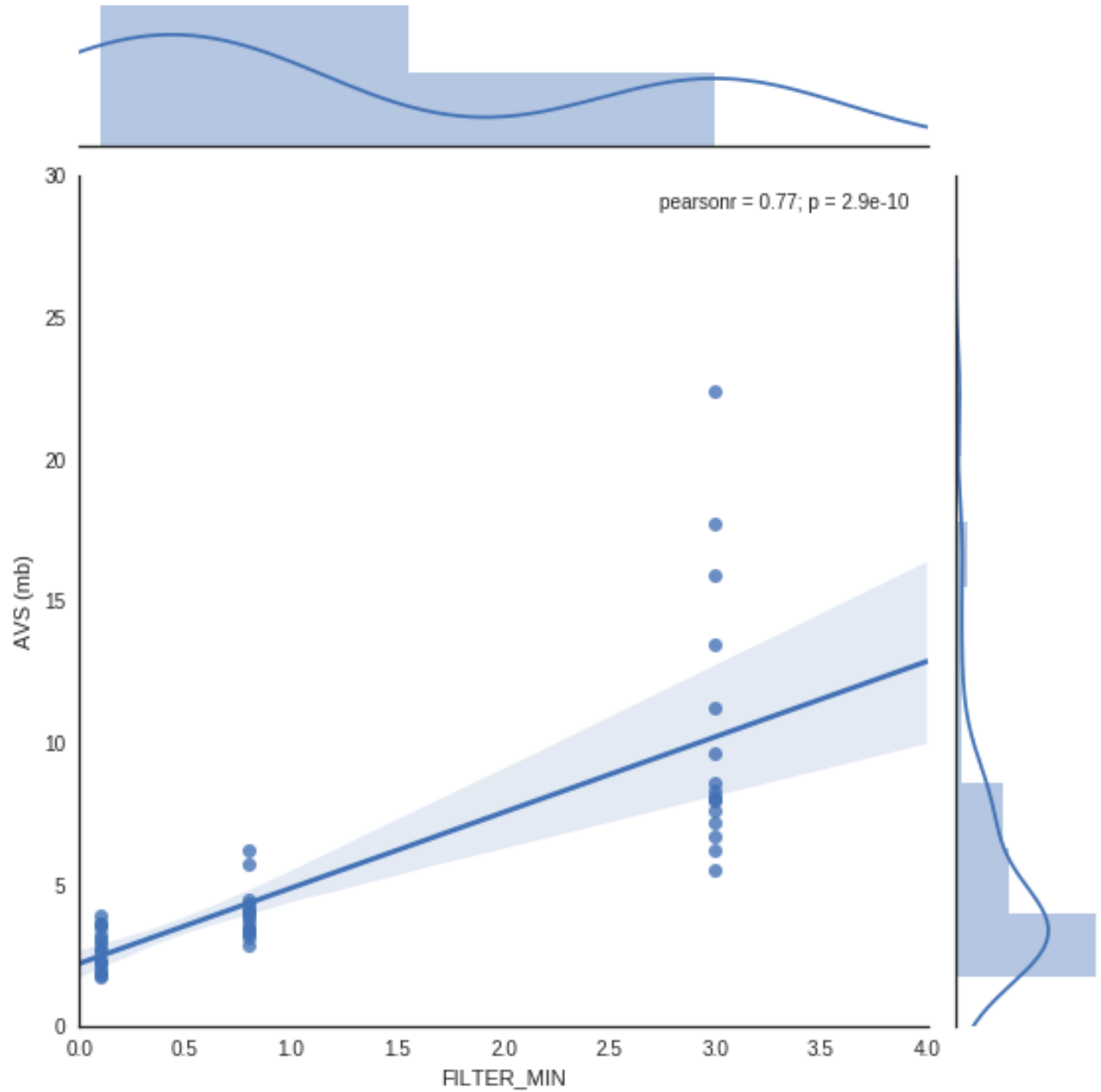
Plots with seaborn

```
sns.set_style("white")
plt.figure(figsize=(10,10))

x=merged["FILTER_MIN"]
y=merged["AVS (mb)"]

sns.violinplot(x,y)
plt.ylabel("AVS (mb)")
plt.xlabel("Groups")
plt.xticks(range(len(x.drop_duplicates())), x.drop_duplicates(),rotation="vertical")
plt.show()
```


AVS (mb)
25
20
15
10
5
0
0.1
0.8
3.0
Groups

Plots with seaborn

```
sns.set_style("white")

plt.figure(figsize=(20,20))

x="FILTER_MIN”

y="AVS (mb)”

sns.jointplot(x=x, y=y,xlim=[0,4], ylim=[0,30],data=merged,size=8,kind="reg")

plt.show()
```


pearsonr = 0.77; p = 2.9e-10
AVS (mb)
FILTER_MIN
0
5
10
15
20
25
30
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Questions ?