Python for Data Science

0.1 Introduction to Python for Data Science

0.1.1 Python basics and lists

(1) Check data type

```
type(a)
```

(2) True and False

```
True + False                # =1
"I said Hey" + ("Hey"*2) # I said Hey HeyHey
```

(3) Convert data type

```
int()   # convert to integer
float() # convert to float
str()   # convert to string
bool()  # convert to boolean: bool(1) = True, bool(2) = True, bool(0) = False
```

(4) Delete element in a list

```
del(fam[2]) # delete 3rd element in list "fam"
```

(5) Copy lists

```
# Reference copy
X = ["a", "b", "c"]
Y = X        # X --> |a|b|c| <-- Y
Y[1] = "z" # change 2nd element of Y will effectively change X
X[1]         # "z", as X, Y point to the same block of memory

# Explicit copy
X = ["a", "b", "c"]
Y = list(X)
```

```
# OR
Y = X[:]
Y[1] = "z"
X[1] # "b", as X, Y point to different blocks of memory
      # X --> |a|b|c|, Y --> |a|z|c|
```

0.1.2 Functions

```
round(1.68, 1) # 1.7
round(1.68)    # 2
help(round)    # round(number [, ndigits]) --> number

?sorted
help(sorted)   # sorted(full, reverse=True)

# Other functions
type(), len(), int(), bool(), float(), str()
```

0.1.3 Methods

```
sister = "liz"
height = 1.73
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.83]

# list methods
fam.index("mom") # 4
fam.count(1.73) # 1
fam.remove("liz") # remove element "liz"
fam.reverse() # reverse all element order
fam.append("me")
fam.append(1.73)

```

```
# string methods
sister.capitalize() # Liz
sister.replace("z", "sa") # lisa
sister.index("z") # 2
sister.count("z") # 1

# Functions vs Methods
# Functions: take in object as argument
type(fam)
# Methods: call functions on objects
fam.index("dad")
```

	type	examples of methods
object	str	capitalize(), replace()
object	float	bit_length(), conjugate()
object	list	index(), count()

0.1.4 Packages: directory of Python scripts

```
pkg/
    mod1.py
    mod2.py
    ...

each script = module
specify functions, methods, types
thousands of packages available
    Numpy (array)
    Matplotlib (data visulization)
    scikit-learn (machine learning)
```

```python

# import packages
import numpy as np
np.array([1, 2, 3])

# if you only want to use array function from numpy
from numpy import array as ary # specific part of the package
array([1, 2, 3]) # not clear if it is from numpy
ary([1, 2, 3])

# pi is in math package
```

0.1.5 NumPy: Numeric Python

(1) Alternative to Python list: NumPy array

(2) Calculation over entire arrays

(3) Easy and fast

```python
# array() takes in list and forms array
height  = [1.73, 1.68, 1.71]
weight = [55, 60, 70]
np_height = np.array(height)
np_weight = np.array(weight)
bmi = np_weight/np_height**2 # calculation over entire arrays

# Remarks
# (1) NumPy arrays: contain only one type ("type coercion")
np.array([1.0, "is", True]) # array(['1.0', 'is', 'True'], dtype='<U32')
# (2) Different types different behavior
python_list = [1, 2, 3]
numpy_array = np.array([1, 2, 3])
python_list + python_list # [1, 2, 3, 1, 2, 3]
```

```
numpy_array+numpy_array # array([2, 4, 6])
# (3) NumPy Subsetting
bmi # array([21.852, 20.975, 21.75, 24.747, 21.441])
bmi[1]       # 20.975
bmi > 23     # array([False, False, False, Ture, False], dtype = bool)
bmi[bmi>23] # array([24.747])

y = bmi>23
bmi[y]
```

0.1.6 2D NumPy

```
np_2d = np.array([1.73, 1.68, 1.71],
                 [65.4, 59.2, 63.6])
np_2d.shape # (2, 3)  2 rows, 5 columns, shape is an attribute

# Subsetting
np_2d[0]     # row 0
np_2d[0][2] # row 0, column 2 --> 1.71
np_2d[0, 2] # [row, column] --> 1.71
```

0.1.7 NumPy: Basic Statistics

```
np.mean(np_city[:,0]) # mean height
np.median(np_city[:,0]) # median
np.corrcoef(np_city[:,0], np_city[:,1]) # array([ [1.0, -0.02], [-0.02, 1.0]
    ])
np.std(np_city[:,0])

# NumPy Array enforces single data type: speed! FASTER.
# Other statistics
sum(), sort(), ...
```

```python

# Generate Data
height = np.round(np.random.normal(1.75, 0.20, 5000), 2) #normal(mean, std, #)
weight = np.round(np.random.normal(60.32, 15, 5000), 2)
np_city = np.column_stack(height, weight)
```

0.2 Intermediate Python for Data Science

0.2.1 Basic plots with matplotlib

```
import matplotlib.pyplot as plt  # subpackage
# (1) Line plot
plt.plot(x,y)
plt.show()

# (2) Histogram
plot.clf()
plt.hist(x,bins=10)

# (3) Scatter plot
plt.scatter(x,y) # asses correlation between 2 variables
plt.scatter(x,y,size,color,alpha=0.8) # alpha = transparency

# Other functions
plt.xscale('log') # put x-axis on a logrithmic scale
plt.grid(True)
plt.text(1550,71,'India') # text(x_axis,y_axis,text)
plt.xlabel('x')
plt.ylabel('y')
plt.title('title')
plt.yticks([0,2,4,6,8,10])
plt.yticks([0,2,4,6,8,10],
           ['0', '2B', '4B', '6B', '8B', '10B'])
```

0.2.2 Dictionary

```
# Method 1: 2 lists, find population of 'alb'
pop = [30, 40, 45]
countries = ['afg', 'alb', 'alg']
```

```
ind = countries.index('alb')
pop[ind]

# Method 2: 1 dictionary
world = {"afg": 30, "alb": 40, "alg": 45}
world["alb"]

# key: value
dict_name[key]   # result: value
dict_name.keys() # get keys
```

```
# Problem
world = {"a":1, "b":2, "c":3, "a":4}
print(world) # {"a":4, "b":2, "c":3}, last one is kept!

# keys have to be "immutable" objects, once created, cannot be changes!!! Strings, booleans, integers and floats are immutable objects, but list is mutable, since you can change it after you create it
{0:"hello", True:"dear", "two":"world"} # correct
{["just","to","test"]:"value", 2:3}     # Type Error: unhashable type: list
```

```
# Add element or update values
world["sealand"] = 0.027
"sealand" in world # True

# Remove element
del(world["sealand"])

# Create dict object
zipped_list = zip(list1, list2) # key = list1, value = list2
rs_dict = dict(zipped_list)
```

List vs Dictionary

List	Dictionary
select, update, and remove: [ ]	select, update, and remove: [ ]
indexed by range of numbers	indexed by unique keys
collection of values order matters select entire subsets	lookup table with unique keys (faster!)

```
# Values of dictionary can be dictionary
europe = {"Spain": {"capital":"madrid", "population":46.77},
          "France": {"capital":"paris", "populatioin":66.03}}
print( europe["Spain"]["population"] ) # 46.77
```

0.2.3 Pandas

Pandas is an open source library, providing high-performance, easy-to-use data structures and data analysis tools for Python.

Tabular dataset:

(1) 2D NumPy array? No, one data type. Columns have different data type!

(2) Pandas!

> high level data manipulation tool

> built on NumPy package

> DataFrame

```
# (1) DataFrame from Dictionary
dict = {
    "country": ["Brazil", "Russia", "India"],
    "capital": ["Brasilia", "Moscow", "New Delhi"],
    "area": [8.516, 17.10, 3.286],
    "population": [200.4, 143.5, 1252]}
import pandas as pd
```

		country (str)	capital (str)	area (float)	population (float)
0	BR	Brazil			
1	RU	Russia			
2	IN	India			
3	CN	China			
4	SA	South Africa			

The red numbers are "iloc"; row labels are "loc".

```
brics = pd.DataFrame(dict) # Pandas assigns automatic row labels: 0, 1, 2,...
brics.index = ["BR", "RU", "IN"] # attribute index

# (2) DataFrame from CSV (Comma Separated Values)
brics = pd.read_csv("brics.csv") # 1st column is set to 0, 1, 2,...
brics = pd.read_csv("brics.csv", index_col=0) # 1st column is the row labels!

# Plot DataFrame data
df_data.plot(kind="scatter", x="Year", y="Population")
# Initialize empty DataFrame
data = pd.DataFrame()
```

0.2.4 Pandas 2

Index and select data:

(1) Square brackets

(2) Advanced methods: loc, iloc

```
# (1) Column Access [ ]
brics["country"]        # return dtype: series
type(brics["country"]) # pandas.core.series.Series: 1D labelled array ==> put
    together a bunch of Series yield DataFrame
```

```
brics[["country"]]            # return dtype: DataFrame
type(brics[["country"]])      # pandas.core.frame.DataFrame
brics[["country","capital"]] # 2 columns with labels given; sub-DataFrame

# (2) Row access [ ]
brics[1:4]

# (3) Element access
# 2D NumPy arrays: my_array[rows, columns]
# Pandas: square brackets: limited functionality (NO!)
# Pandas: loc (label-based), iloc (integer position-based)

# loc and iloc
# (i) row access
brics.loc["RU"]   # row as Pandas Series
brics.loc[["RU"]] # row as DataFrame
brics.loc[["RU","IN","CH"]] # multiple rows
brics.iloc[[1,2,3]]
# (ii) row and column access
brics.loc[["RU","IN","CH"],["country","capital"]]
brics.loc[:,["country","capital"]]
brics.iloc[:,[0,1]]
brics.iloc[[1,2,3],[0,1]]
```

```
# Recap
brics.head() # head of DataFrame
brics.tail() # tail of DataFrame

# (I) Square brackets
brics[["country","capital"]] # column access
```

```
brics[1:4] # row access

# (II) loc (label-based) and iloc (integer position-based)
brics.loc[["RU","IN","CH"]] # row access
brics.iloc[[1,2,3]]          # row access
brics.loc[:,["country","capital"]] # column access
brics.iloc[:,[0,1]]                # column access
brics.loc[["RU","IN","CH"],["country","capital"]] # row and column access
brics.iloc[[1,2,3],[0,1]]                         # row and column access
```

0.2.5 Logic, Control (Flow and Filtering)

```
print(True=1) # True
print(False=0) # True
print(False=-1) # False
print(True>False) # True
```

```
# (1) operational operators
>, <, >=, <=, ==, !=

# (2) boolean operators
and, or, not
logical_and(), logical_or(), logical_not()  # for NumPy array
np.logical_and(bmi>21, bmi<22)
bmi[np.logical_and(bmi>21, bmi<22)]

# (3) conditional statements
if, else, elif
```

```
# Filtering DataFrame
# (1) operational operators
is_huge = brics["area"]>8 # Series
```

```
brics[is_huge]
brics[ brics["area"]>8 ]
# (2) boolean operators
np.logical_and(brics["area"]>8, brics["area"]<10)
brics[ np.logical_and(brics["area"]>8, brics["area"]<10) ]
brics[ brics[ np.logical_and(brics["area"]>8, brics["area"]<10) ] ]
```

0.2.6 Loop

```
# Dictionary is inherently unordered.
```

```
# while loop
while condition:
    statement
# for loop
for var in seq:
    expression
```

```
# (1) Loop over list and string
for height in fam:
    print(height)
for index, height in enumerate(fam):
    print("index" + str(index) + ":" + str(height))
for c in "family":
    print(c.capitalize())

# (2) Loop over dictionary and NumPy array
# dictionary
world = {"agf":30.55, "alb":2.77, "alg":39.21}
for key, value in world.items()
    print(key + "--" + str(value))
# 1-D NumPy array
bmi = np.array([10, 15, 20])
```

```
for val in bmi:
    print(val)
# 2-D NumPy array
height = [1.73, 1.68, 1.71]
weight = [65.4, 59.2, 63.6]
meas = np.array([height, weight])
for val in meas:
    print(val) # print entire array --> [1.73,1.68,1.71], [65.4,59.2,63.6]
for val in np.nditer(meas):
    print(val) # print 1st column (height), then 2nd column (weight)

# Recap
# Dictionary
for key, value in my_dict.items(): # method
    ...
# NumPy array
for val in np.nditer(my_array): # function
    ...

# (3) Loop over Pandas DataFrame
for val in brics:
    print(val)
# output is column names one by one
"""
country
capital
area
population
"""
for lab, row in brics.iterrows():
    print(lab)
```

```
    print(row)
# output:
"""
BR
country Brazil
capital ...
area ...
population ...
Name: BR, dtype: object.
RU
country Russia
capital ...
area ...
population ...
Name: RU, dtype: object.
...
"""
for lab, row in brics.iterrows():
    brics.loc[lab,"name_length"]=len(row["country"])] # add a new column
# The above creates Series on every iteration (row is a Series), better way is the following
brics["name_length"] = brics["country"].apply(len) # function
cars["country"] = cars["country"].apply(str.upper) # method
```

0.2.7 Case Study: Hacker Statistics

random: a sub-package of NumPy

Pseudo-random numbers from computer (same seed generates same “random” numbers): reproducibility.

```
np.random.seed(123)
coin = np.random.randint(0,2) # [0,1], 2 is not included
```

	country	capital	area	population
BR	Brazil			
RU	Russia			
IN	India			
CN	China			
SA	South Africa			

```
float_num = np.random.rand() # float numbers

# count number of values in "var" that are great than 60
sum(var[var>60])
```

0.3 Python Data Science Toolbox I

0.3.1 Tuples

(1) like a list—can contain multiple values

(2) immutable—can't modify values!

(3) constructed using parentheses ()

```
even_numbers = (2, 4, 6)
```

(4) unpack a tuple into several variables in one line

```
a, b, c = even_numbers
```

(5) access tuple elements like you do with lists

```
second_num = even_numbers[1]
```

(6) uses zero-indexing

0.3.2 Scope

(1) global scope—defined in the main body of a script

(2) local scope—defined inside a function

(3) built-in scope—names in pre-defined built-ins module

```
new_val = 10 # new_val here is global
def square(val):
    new_val = value**2 # new_value here is local, and ceases to exist outside the function
    return new_val

square(3) # 9
new_val # 10
```

```python
new_val = 10 # new_val here is global
def square(val):
    new_val2 = new_val**2
    return new_val2

square(3) # 100
new_val = 20
square(3) # 400
```

```python
"""Python first searches within local scope, then global scope, lastly built-in scope."""
```

```python
"""### Change/alter a global variable within a function ###"""
new_val = 10
def square(value):
    global new_val
    new_val = new_val**2
    return new_val

square(3) # 100
new_val # 100
```

```python
"""Python built-in scope: a built-in module "builtins" """
import builtins
dir(builtins) # list all the names in module "builtins"
```

0.3.3 Nested Functions

(1) Avoid writing out the same computations within functions repeatedly

```python
def mod2plus5(x1, x2, x3): # enclosing scope
    """ Return the remainder plus 5 of three values """
    def inner(x): # local scope
```

```python
        return x % 2 + 5
    return (inner(x1), inner(x2), inner(x3))
```

```python
def outer(...): # <--- enclosing function
    """..."""
    x = ...
    def inner(...): # <-- local scope
        """..."""
        y = x**2
    return ...

"""
Python searches the local scope of the function inner(), if not found, then searches the scope of enclosing function outer(), then global scope, lastly build-in scope.
Scopes searched: local scope --> enclosing functions --> global --> built-in
"""
```

(2) The idea of closure (returning functions): the nested function/inner function remembers the state of its enclosing scope when called; thus, anything defined locally in the enclosing scope is available to the inner function even when the outer function has finished execution.

```python
def raise_val(n):
    """Return the inner function"""
    local = 3
    def inner(x):
        """Raise x to the power of n"""
        raised = x ** n + local
        return raised
    return inner

square = raise_val(2)
```

```python
cube = raise_val(3)
print(square(5), cube(4)) # 25+3, 64+3
```

```python
"""### Create/change variable in an enclosing scope for the nested function
    ###"""
def outer():
    """Print the value of n"""
    n = 1
    def inner():
        nonlocal n
        n = 2
        print(n)
    inner()
    print(n)
outer() # 2, 2
"""Cannot use global in enclosing function, then use nonlocal within the inner
    function on the same variable"""
```

0.3.4 Default and flexible arguments

```python
(1) Default argument
def power(number, pow=1):
    new_value = number ** pow
    return new_value

power(9, 2) # 81
power(9, 1) # 9
power(9) # 9
```

```python
(2.1) Flexible arguments: *args
def add_all(*args):
    sum_all = 0
```

```
    for num in args:
        sum_all += num
    return sum_all

(2.2) Flexible arguments: **kwargs (dictionary)
def print_all(**kwargs):
    for key, value in kwargs.items():
        print(key + ": " + value)
```

0.3.5 Lambda functions and error-handling

```
(1) Lambda Functions
raise_to_power = lambda x, y: x ** y # fcn_name = lambda input: output
raise_to_power(2,3) # 8

"Anonymous functions: the best use of lambda functions are for when you want simple functionalities to be anonymously embedded within larger expressions. (The function is not stored in the environment, unlike a function defined with def.)"

# (1.1) map(func, seq)
Pass lambda functions to map without naming them; they are called anonymous fucntions
  Function map takes two arguments: lambda function "func" and list "seq"
  map() applies the function to ALL elements in the sequence

nums = [48, 6, 9, 21, 1]
square_all = map(lambda num: num ** 2, nums) # generate a map object
print(list(square_all)) # turn map to list [2304, 36, 81, 441, 1]

# (2) filter(func, seq)
fellowship = ["frodo", "samwise", "merry", "aragorn", "legolas", "boromir"]
```

```
result = filter(lambda member: len(member)>6, fellowship)
print(list(result)) # ["samwise", "aragorn", "legolas", "boromir"]

# (3) reduce(func, seq)  # Perform computation on list, return a single value
from functools import reduce
stark = ["robb", "sansa", "arya"]
result = reduce(lambda item1, item2: item1+item2, stark)
print(result) # robbsansaarya
```

```
(2) Error Handling
Errors and exceptions
> Exceptions——caught during execution
> Catch exceptions with try-except clause
    - runs the code following try
    - if there is an exception, run the code following except

def sqrt(x):
    """Return the square root of a number"""
    try:
        return x**0.5
    except:
        print("x must be an int or float")

def sqrt(x):
    """Return the square root of a number"""
    try:
        return x**0.5
    except TypeError: # only catches type error but let other errors through
        print("x must be an int or float")

def sqrt(x): # instead of "printing an error", we "raise an error"!
```

```python
    """Return the square root of a number"""
    if x<0:
        raise ValueError("x must be non-negative")
    try:
        return x**0.5
    except TypeError:
        print("x must be an int or float")
```

0.4 Python Data Science Toolbox II

0.4.1 Iterators vs Iterables

```
Iterable:
    E.g.: lists, strings, dictionaries, file connections, range objects
    An object with an associated iter() method
    Applying iter() to an iterable creates an iterator

Iterator:
    An object with an associated next() method (produces next value)

ps: range object, range(10**100) does not give any error, instead it creates a
    range object but does not precreate a list object.
```

```
# Create iterator from an iterable
"(1) iterating with next()"
word = "Da"
it = iter(word)
next(it) # "D"
next(it) # "a"
next(it) # throw an iterator error

"(2) iterating once with *"
word = "Data"
it = iter(word)
print(*it) # D a t a
print(*it) # No more values to go through

"(3) iterating over dictionaries"
mydict = {"a": 1, "b": 2}
for key, value in mydict.item():
```

```
    print(key, value)
# a 1
# b 2

"(4) iterating over file connections"
file = open("file.txt")
it = iter(file)
print(next(it)) # this is the 1st line
print(next(it)) # this is the 2nd line
```

```
Summary
An iterable is an object that can return an iterator
An iterator is an object that keeps state and produces the next value when you
    call next() on it

PS: pass iterator from range() to list() and sum()
values = range(10, 21)
values_list = list(values)
values_sum = sum(values)
```

0.4.2 enumerate() and zip()

```
(1) enumerate is a function that takes in any iterable as argument, and
    returns a special enumerate object, which consists of pairs, containing
    the element of the original iterable, along with their index within the
    iterable.

iterable = ["a", "b", "c"]
enumerate_obj = enumerate(iterable) # enumerate_obj is also an iterable
tuple_list = list(enumerate_obj)
print(tuple_list) # [(0, "a"), (1, "b"), (2, "c")]

```

```
for index, value in enumerate(iterable):
    print(index, value) # 0 a, 1 b, 2 c
for index, value in enumerate(iterable, start=10)
    print(index, value) # 10 a, 11 b, 12 c
```

```
(2) zip is a function that accepts an arbitrary number of iterables, and
    returns an iterator of tuples

list1 = ["1", "2", "3"]
list2 = ["a", "b", "c"]
zip_obj = zip(list1, list2)
print(*zip_obj) # ("1", "a") ("2", "b") ("3", "c")
tuple_list = list(zip_obj)
print(tuple_list) # [ ("1", "a"), ("2", "b"), ("3", "c")]
for z1, z2 in zip(list1, list2):
    print(z1, z2) # 1 a, 2 b, 3 c

unzip a zip object by using * within zip()
zip_obj = zip(list1, list2)
list3, list4 = zip(*zip_obj)
```

0.4.3 Using iterators to load large files into memory

```
Loading data in chunks
    There can be too much data to hold in memory
    Solution: load data in chunk!
    Pandas function: read_csv() --> specify the chunk: chunksize

# iterating over data
import pandas as pd
result = []
for chunk in pd.read_csv("data.csv", chunksize = 1000):
```

```
    result.append(sum(chuck["x"]))
total = sum(result)

total = 0
for chunk in pd.read_csv("data.csv", chunksize = 1000):
    total += sum(chunk["x"])
```

0.4.4 List Comprehensions

```
(1.1) For loop vs list comprehension
new_nums = []
for num in nums:
    new_nums.append(num+1)

# output expr: num+1; iterator var: num; iterable: nums
new_nums = [num+1 for num in nums]
result = [num for num in range(11)]

Summary
list comprehensions:
    Collapse for loops for building lists into a single line
    Components: iterable, iterator variables (represent members of iterables),
     output expression
```

```
(1.2) Nested loops
For loop vs list comprehension
Example 1:
pairs = []
for num1 in range(0, 2):
    for num2 in range(6, 8):
        pairs.append(num1, num2)
print(pairs) # [(0,6), (0,7), (1,6), (1,7)]
```

```

pairs = [(num1, num2) for num1 in range(0,2) for num2 in range(6,8)]

Example 2:
matrix = [ [0,1,2,3,4],
           [0,1,2,3,4],
           [0,1,2,3,4],
           [0,1,2,3,4],
           [0,1,2,3,4]]
for row in range(0,5):
    row = []
    for col in rane(0,5):
        row.append(col)
    matrix.append(row)

matrix = [ [col for col in range(0,5)] for row in range(0,5)]
```

```
(2.1) Conditionals in comprehensions
# conditionals on iterable
[num**2 for num in range(10) if num%2 == 0]  # [0, 4, 16, 36, 64]
# conditionals on output expression
[num**2 if num%2 == 0 else 0 for num in range(10)] # [0,0,4,0,16,0,36,0,64,0]
```

```
(2.2) Dict comprehensions {}
pos_nge = {num:-num for num in range(3)} # {0:0, 1:-1, 2:-2}
```

0.4.5 Generator Expressions

```
Range objects and generator:
range_obj = range(10000000000)
generator_obj = (num for num in range(1000000000))

```

```
"Lazy evaluation: the evaluation of the expression is delayed until its values is needed."
[num for num in range(10**1000000)] # ERROR, not enough memory
(num for num in range(10**1000000)) # OK, no construction/storage in memory
```

List Comprehension	Generator
uses []	uses ()
creates list obj	creates generator obj
stores list in memory	does not store/construct list in momory
can be iterated over	can be iterated over: result = (num for num in range(3)) for num in result: print(num) # 0 1 2
—	passes it to list –> get list, e.g. list(generator)
— iter()	passes it to next –> get elem, e.g. next(generator)

```
Generator function (yield) # yields generator object
def num_sequence(n):
    """Generates values from 0 to n"""
    i = 0
    while i<n:
        yield i
        i += 1

Other generators: dict.items(), range()
```

```
Re-cap: list comprehensions
Basic
  [output_expr  for iterator_var in iterable]
Advanced
```

```
[output_expr conditional_on_output for iterator_var in iterable conditional_on_iterable]
```

0.4.6 Context Manager

```
"Ensures that resources are efficiently allocated when opening a connection to a file"
# Open a connection to the file
with open("world_dev_ind.csv") as file: # file is file obj == generator
    # Skip the column names
    file.readline()
    # Initialize an empty dictionary
    counts_dict = {}
    # Process only the first 1000 rows
    for j in range(1000):
        # Split the current line into a list: line
        line = file.readline().split(', ')
        if not line:
            break # reaches end of file
        # Get the value for the first column: first_col
        first_col = line[0]
        # If the column value is in the dict, increment its value
        if first_col in counts_dict.keys():
            counts_dict[first_col] += 1
        else:
            counts_dict[first_col] = 1
# Print the resulting dictionary
print(counts_dict)

# generate reader object, use next() to read chunk by chunk
pd.read_csv(file_name, chunksize=100)
```