Durham University

Topic 3：Pandas

Week 3/4

jingyun.wang@durham.ac.uk

https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html
https://www.datacamp.com/community/tutorials/python-excel-tutorial

Pandas

- Is built on top of Numerical Python (popularly known as NumPy).
- uses its multi-dimensional arrays and fast operations internally to provide higher level methods for manipulation and analysis.
- Almost every Pandas method returns a (modified) copy of the data, which allows you to chain transformations, and perform complex modifications in one line.

Disadvantages :

it can be slower in loading, reading, and analyzing big datasets with millions of records.

Durham University

Data Structures

Pandas' data structures can hold mixed typed values as well as labels, and their axes can have names set.

Series: basically a 1-dimensional labeled array, have only one axis (axis == 0) called “index”.

```
import pandas as pd
import numpy as np
```

```
pd.Series([7, 90, 'durham', np.nan])
```

```
0         7
1        90
2    durham
3       NaN
dtype: object
```

```
pd.Series([7, 90, 'durham', np.nan], index=['a', 'B', 'C', 'd'])
```

```
a         7
B        90
C    durham
d       NaN
dtype: object
```

pandas primarily uses the value np.nan to represent missing data. It is by default not included in computations.

String Methods

Series is equipped with a set of string processing methods in the str attribute that make it easy to operate on each element of the array, as in the code snippet below. Note that pattern-matching in str generally uses regular expressions by default (and in some cases always uses them).

```
s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"])

s.str.lower()
```

```
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

DataFrames: can be thought of as Python dictionaries where the keys are the column labels, and the values are the column Series.

```
pd.DataFrame({'day': [11, 30], 'month': [1, 12], 'year': [2010, 2021]})
```

	day	month	year
0	11	1	2010
1	30	12	2021

```
df = pd.DataFrame(
 {"A": 1.0,
  "B": pd.Timestamp("20130102"),
  "C": pd.Series(1, index=list(range(4)), dtype="float32"),
  "D": np.array([3] * 4, dtype="int32"),
  "E": pd.Categorical(["test", "train", "test", "train"]),
  "F": "foo"
 })
df
```

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

```
df.head(3)
```

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo

```
df.tail(2)
```

	A	B	C	D	E	F
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

Panels: 3-dimensional data structures, rarely used

Analogously to DataFrames, they can be thought of as Python dictionaries of DataFrames. Instead of “index” and “columns”, Panels’ axes are named as follow:

items (axis == 0)

major_axis (axis == 1)

minor_axis (axis == 2)

Durham University

DataFrames

Viewing data--describe() shows a quick statistic summary of your data:

```
df.describe()
```

	A	C	D
count	4.0	4.0	4.0
mean	1.0	1.0	3.0
std	0.0	0.0	0.0
min	1.0	1.0	3.0
25%	1.0	1.0	3.0
50%	1.0	1.0	3.0
75%	1.0	1.0	3.0
max	1.0	1.0	3.0

Descriptive Statistics in Python Pandas

Sr.No.	Function	Description
1	count()	Number of non-null observations
2	sum()	Sum of values
3	mean()	Mean of Values
4	median()	Median of Values
5	mode()	Mode of values
6	std()	Standard Deviation of the Values
7	min()	Minimum Value
8	max()	Maximum Value
9	abs()	Absolute Value
10	prod()	Product of Values
11	cumsum()	Cumulative Sum
12	cumprod()	Cumulative Product

Note – Since DataFrame is a Heterogeneous data structure. Generic operations don't work with all functions.

Functions like sum(), cumsum() work with both numeric and character (or) string data elements without any error. Though n practice, character aggregations are never used generally, these functions do not throw any exception.

Functions like abs(), cumprod() throw exception when the DataFrame contains character or string data because such operations cannot be performed.

Example:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

df.mean()

Performing a descriptive statistic:

```
df.mean()
```

```
A    1.0
C    1.0
D    3.0
dtype: float64
```

Same operation on the other axis:

```
df.mean(1)
```

```
0    1.666667
1    1.666667
2    1.666667
3    1.666667
dtype: float64
```

sum()

Durham University

Viewing data-- Transposing your data:

```
df.T
```

	0	1	2	3
A	1	1	1	1
B	2013-01-02 00:00:00	2013-01-02 00:00:00	2013-01-02 00:00:00	2013-01-02 00:00:00
C	1	1	1	1
D	3	3	3	3
E	test	train	test	train
F	foo	foo	foo	foo

Viewing data-- Sorting by an axis:

```
df.sort_index(axis=1, ascending=False)
```

	F	E	D	C	B	A
0	foo	test	3	1.0	2013-01-02	1.0
1	foo	train	3	1.0	2013-01-02	1.0
2	foo	test	3	1.0	2013-01-02	1.0
3	foo	train	3	1.0	2013-01-02	1.0

Viewing data-- Sorting by values:

```
df.sort_values(by="E")
```

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
2	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
3	1.0	2013-01-02	1.0	3	train	foo

Selection

Selection by label
Selection by position
Boolean indexing

Selecting a single column, which yields a Series, equivalent to df.E:

```
df["E"]
```

```
0     test
1    train
2     test
3    train
Name: E, dtype: category
Categories (2, object): [test, train]
```

Selecting via [], which slices the rows.

```
df[0:3]
```

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo

Durham University

Merging

https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html#merging

https://towardsdatascience.com/3-key-differences-between-merge-and-concat-functions-of-pandas-ab2bab224b59#:~:text=Concat%20function%20concatenates%20dataframes%20along,combinations%20based%20on%20a%20condition.

https://towardsdatascience.com/pandas-join-vs-merge-c365fd4fbf49

Merge, join, concatenate and compare

pandas provides various facilities for easily combining together Series and DataFrame objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

Concatenating objects

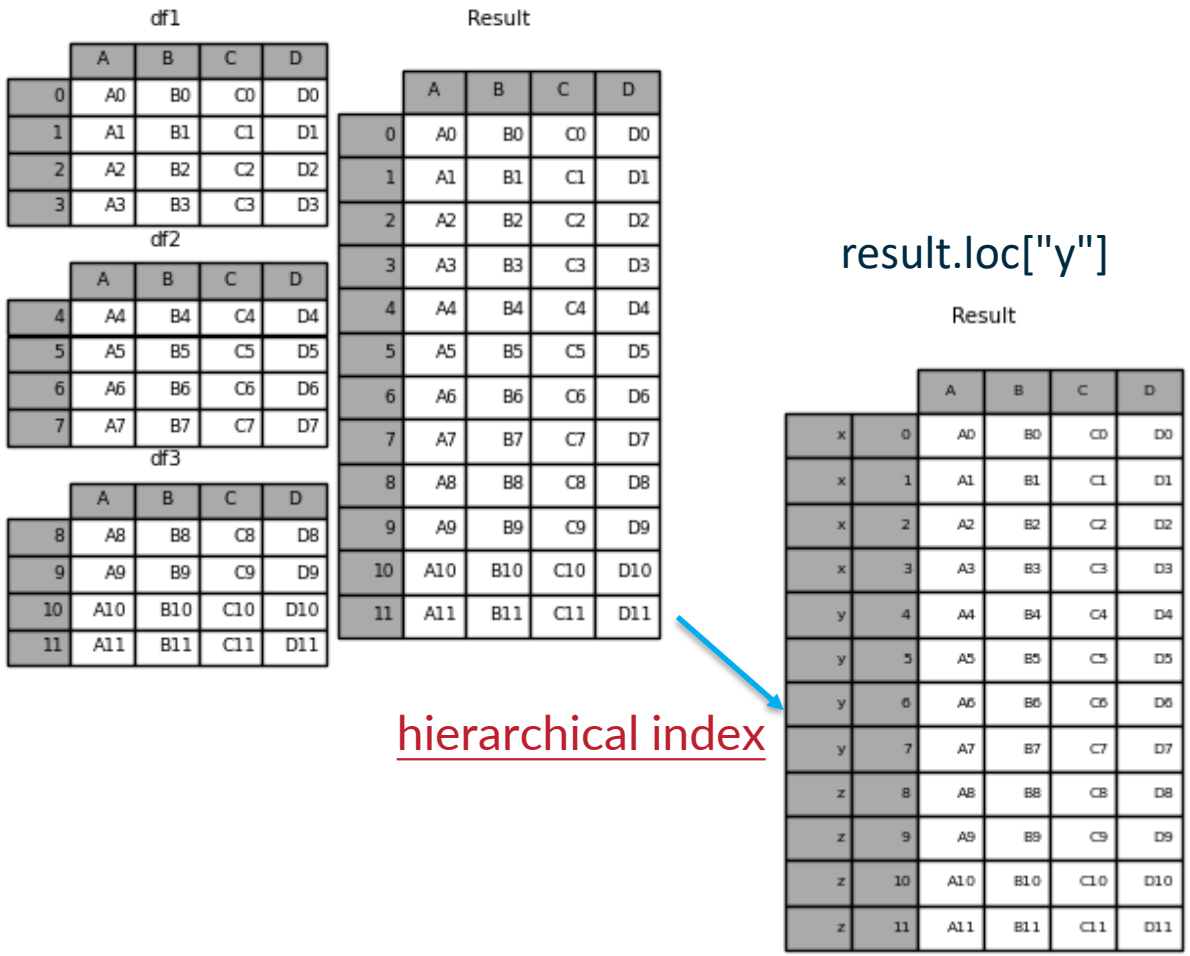
```
df1 = pd.DataFrame(
    {
        "A": ["A0", "A1", "A2", "A3"],
        "B": ["B0", "B1", "B2", "B3"],
        "C": ["C0", "C1", "C2", "C3"],
        "D": ["D0", "D1", "D2", "D3"],
    },
    index=[0, 1, 2, 3],
)


df2 = pd.DataFrame(
    {
        "A": ["A4", "A5", "A6", "A7"],
        "B": ["B4", "B5", "B6", "B7"],
        "C": ["C4", "C5", "C6", "C7"],
        "D": ["D4", "D5", "D6", "D7"],
    },
    index=[4, 5, 6, 7],
)


df3 = pd.DataFrame(
    {
        "A": ["A8", "A9", "A10", "A11"],
        "B": ["B8", "B9", "B10", "B11"],
        "C": ["C8", "C9", "C10", "C11"],
        "D": ["D8", "D9", "D10", "D11"],
    },
    index=[8, 9, 10, 11],
)


frames = [df1, df2, df3]

result = pd.concat(frames)
```



associate specific keys with each of the pieces of the chopped up DataFrame by using the keys argument:

result = pd.concat(frames, keys=["x", "y", "z"])

Durham University

concat () :join

When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following two ways:

Take the union of them all, join='outer'. This is the default option as it results in zero information loss.

```
df4 = pd.DataFrame(
    {
        "B": ["B2", "B3", "B6", "B7"],
        "D": ["D2", "D3", "D6", "D7"],
        "F": ["F2", "F3", "F6", "F7"],
    },
    index=[2, 3, 6, 7],
)

result = pd.concat([df1, df4], axis=1)
```

#Take the intersection, join='inner'.

result = pd.concat([df1, df4], axis=1, join="inner")

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Result

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Durham University

pandas.DataFrame.reindex

reuse the exact index from the original DataFrame:

```
In [11]: result = pd.concat([df1, df4], axis=1).reindex(df1.index)
```

```
pd.concat([df1, df4.reindex(df1.index)], axis=1)
```

By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Concatenating using append

A useful shortcut to concat() are the append() instance methods on Series and DataFrame. These methods actually predated concat. They concatenate along axis=0, namely the index:

```
result = df1.append(df2)
```

append may take multiple objects to concatenate:

```
result = df1.append([df2, df3])
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

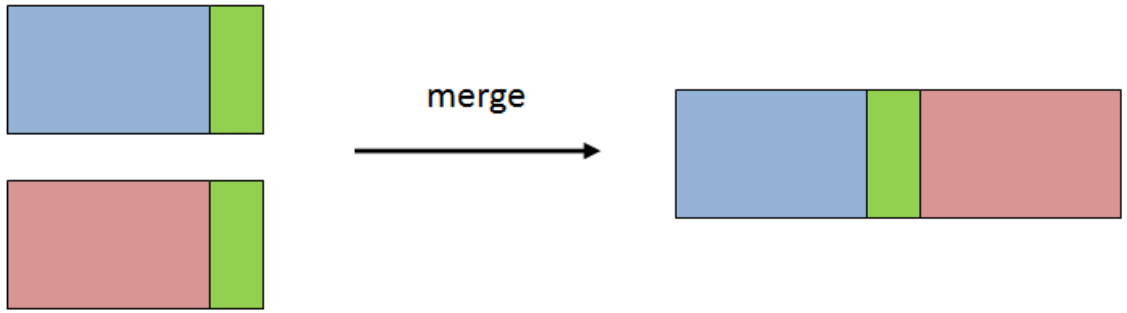
	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

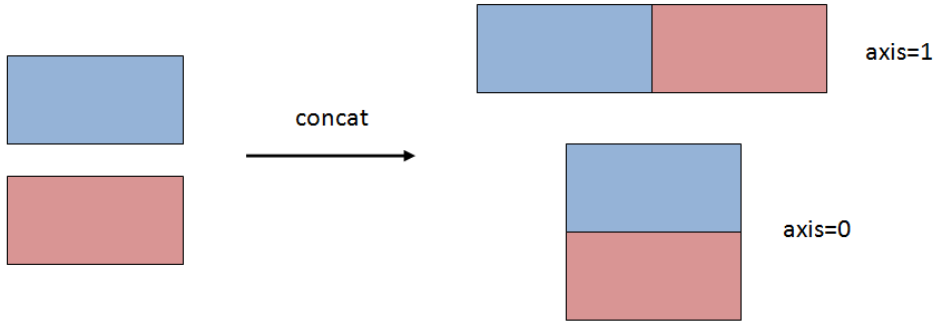
Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

merge()

Merge combines dataframes based on *values in shared columns*. Merge function offers more flexibility compared to concat function because it allows combinations based on a condition.





Concat function concatenates dataframes along rows or columns.

merge():how

The how parameter of the merge function works in a similar way as join.

- inner: only rows with same values in the column specified by on parameter (default value of how parameter)
- outer: all the rows
- left: all rows from left DataFrame
- right: all rows from right DataFrame

First DataFrame

column_a	column_b	column_c

Second DataFrame

column_a	column_b	column_c

how = 'inner'

column_a	column_b_x	column_c_x	column_b_y	column_c_y

how = 'left'

column_a	column_b_x	column_c_x	column_b_y	column_c_y
			NaN	NaN
			NaN	NaN

how = 'outer'

column_a	column_b_x	column_c_x	column_b_y	column_c_y
			NaN	NaN
			NaN	NaN
	NaN	NaN		
	NaN	NaN		

how = 'right'

column_a	column_b_x	column_c_x	column_b_y	column_c_y
	NaN	NaN		
	NaN	NaN		

Durham University

examples

df1

	A	B	C
0	1	True	C1
1	2	False	C2
2	3	True	C3
3	4	True	C4

df2

	A	B	C
2	5	False	C1
3	7	False	C3
4	8	True	C5
5	5	False	C8

```
df1.merge(df2, on='C', how='left')
```

	A_x	B_x	C	A_y	B_y
0	1	True	C1	5.0	False
1	2	False	C2	NaN	NaN
2	3	True	C3	7.0	False
3	4	True	C4	NaN	NaN

```
df1.merge(df2, on='C', how='inner')
```

	A_x	B_x	C	A_y	B_y
0	1	True	C1	5	False
1	3	True	C3	7	False

```
df1.merge(df2, on='C', how='outer')
```

	A_x	B_x	C	A_y	B_y
0	1.0	True	C1	5.0	False
1	2.0	False	C2	NaN	NaN
2	3.0	True	C3	7.0	False
3	4.0	True	C4	NaN	NaN
4	NaN	NaN	C5	8.0	True
5	NaN	NaN	C8	5.0	False

Durham University

Comparing objects

The compare() and compare() methods allow you to compare two DataFrame or Series, respectively, and summarize their differences.

This feature was added in V1.1.0.

Durham University

Working with Excel Files

https://www.datacamp.com/community/tutorials/python-excel-tutorial

Writing to an excel file:

```
df.to_excel("foo.xlsx", sheet_name="Sheet1")
```

Read an excel file:

```
pd.read_excel("foo.xlsx", "Sheet1", index_col=None, na_values=["NA"])
```

Write Pandas DataFrames to Excel Files

practical 2-1-1 xlswriter

```
import xlsxwriter

 # Create a workbook and add a worksheet.
workbook = xlsxwriter.Workbook('students.xlsx')
worksheet = workbook.add_worksheet()

 # Add a bold format to use to highlight cells.
bold = workbook.add_format({'bold': True})

 # Write some data headers.
worksheet.write('A1', 'student ID', bold)
worksheet.write('B1', 'gender', bold)
worksheet.write('C1', 'major', bold)


 # Some data we want to write to the worksheet.
data1 = (
     ['cs000234', 'female','computer science'],
     ['bs101977', 'male','business'],
     ['ds298387',  'female','data science'],
 )

 # Start from the first cell below the headers.
row = 1
col = 0

 # Iterate over the data and write it out row by row.
for sid, gender, major in (data1):
    worksheet.write(row, col,     sid)
    worksheet.write(row, col + 1, gender)
    worksheet.write(row, col + 2, major)
    row += 1

workbook.close()
```

```
import pandas as pd
import xlsxwriter

# Specify a writer
writer = pd.ExcelWriter('students.xlsx', engine='xlsxwriter')

# prepare the dataframe
data1 = pd.DataFrame( {
     'student ID':['cs000234', 'bs101977','ds298387'],
     'gender':['female', 'male','female'],
     'major':['computer science', 'business' ,'data science']
} )

#  write a dataframe as an excel file
data1.to_excel(writer, 'Sheet1')

# Save the result
writer.save()
```

Load Excel Files As Pandas DataFrames

```
# Assign spreadsheet filename to `file`
file = 'students.xlsx'

# Load spreadsheet
xl = pd.ExcelFile(file)

# get the sheet names
sheet=xl.sheet_names
# Print the sheet names
print(xl.sheet_names)

# Load a sheet into a DataFrame by name: df1
#df1 = xl.parse(sheet)
df1 = pd.read_excel(xl,sheet)
print (df1)
```

```
['Sheet1']
OrderedDict([('Sheet1',   student ID  gender             major
0   cs000234  female  computer science
1   bs101977    male          business
2   ds298387  female      data science)])
```

Durham University

practical 2-1-2

```python
import pandas as pd
import xlsxwriter

# Assign spreadsheet filename to `file`
file = 'students.xlsx'

# Load spreadsheet
xl = pd.ExcelFile(file)

# get the sheet names
sheet=xl.sheet_names

# Load a sheet into a DataFrame by name: df1
df1 = pd.read_excel(xl,sheet)

# prepare the dataframe d2
df2 = pd.DataFrame( {
     'student ID':['cs000124', 'bs202933'],
     'gender':['male', 'female'],
     'major':['computer science',  'business']
} )

#df3=df1.append(df2)
df3=pd.concat([df1,df2])
# Specify a writer
writer = pd.ExcelWriter('students2.xlsx', engine='xlsxwriter')
df3.to_excel(writer,'sheet1')
writer.save()
```

```python
import pandas as pd
import xlsxwriter

# Assign spreadsheet filename to `file`
file = 'students.xlsx'

# Load spreadsheet
xl = pd.ExcelFile(file)

# get the sheet names
#sheet=xl.sheet_names

# Load a sheet into a DataFrame by name: df1
#df1 = pd.read_excel(xl,sheet) will return orderedDict instead of d
df1 = pd.read_excel(xl)
# prepare the dataframe d2
df2 = pd.DataFrame( {
     'student ID':['cs000124', 'bs202933'],
     'gender':['male', 'female'],
     'major':['computer science',  'business']
} )

#df3=df1.append(df2)
df3=pd.concat([df1,df2])
# Specify a writer
writer = pd.ExcelWriter('students2.xlsx', engine='xlsxwriter')
df3.to_excel(writer,'sheet1')
writer.save()
```

```
TypeError: cannot concatenate object of type "<class 'collections.OrderedDict'>"; only pd.Series, pd.DataFrame, and pd.Panel (d
eprecated) objs are valid
```

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Durham University

Durham University

Working with CSV Files

https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_spreadsheets.html?highlight=csv#csv

you can read the .csv file using read_csv

```
df = pd.read_csv("example.csv") # Load csv
```

The pd.read_csv() function has a **sep** argument which acts as a delimiter that this function will take into account is a comma or a tab, by default it is set to a comma, but you can specify an alternative delimiter if you want to.
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

you can also write the data frame results back to a comma-separated file using the pandas to_csv() method as shown below:

```
df.to_csv("example.csv")
```

If you want to save the output in a tab-separated fashion, all you need to do is pass a \t to the sep argument.