error handling; pandas and data analysis

Ben Bolker

26 November 2019

## generating errors

* we’ve already seen the raise keyword, in passing
* raise Exception is the simplest way to have your program stop when something goes wrong
* in a notebook/console environment, it stops the current cell/function (doesn’t crash the session)

raise Exception

Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
Exception

* you have to raise <something>
* Exception is the most general case (“something happened”)
* other possibilities
  + TypeError: some variable is the wrong type
  + ValueError: some variable is the right type but the wrong value

x = -1  
if not isinstance(x,str): ## check if x is a str  
 raise TypeError

Traceback (most recent call last):  
 File "<stdin>", line 2, in <module>  
TypeError

import math  
x = -1  
if x<0:  
 raise ValueError  
print(math.sqrt(x))

Traceback (most recent call last):  
 File "<stdin>", line 2, in <module>  
ValueError

## error messages

* it’s always better to be more specific about the cause of an error:

x = -1  
if not isinstance(x,str): ## check if x is a str  
 errstr = "x is of type "+type(x).\_\_name\_\_+", should be str"  
 raise TypeError(errstr)

TypeError: x is of type int, should be str

**f-strings** are a convenient way to construct error messages: anything inside curly brackets is interpreted as a Python expression. e.g.

x=1  
print(f"x is of type {type(x).\_\_name\_\_}, should be str")

## x is of type int, should be str

So we could use

if not isinstance(x,str): ## check if x is a str  
 raise TypeError("x is of type {type(x).\_\_name\_\_}, should be str")

x = -1  
if x<0:  
 raise ValueError(f"x should be non-negative, but it equals {x}")

ValueError: x should be non-negative, but it equals -1

## warnings

An error means “it’s impossible to continue” or “you shouldn’t continue without fixing the problem”. You might want to issue a *warning* instead. This is not too different from just using print(), but it allows advanced users to decide if they want to suppress warnings.

import warnings

warnings.warn("something bad happened")

## <string>:1: UserWarning: something bad happened

## handling errors

Now suppose you are getting an error and you don’t want your program to stop. “Wrapping” your code in a try: clause will allow you to specify what to do in this case. pass is a special Python statement called a “null operation” or a “no-op”; it does nothing except keep going.

try:  
 x= math.sqrt(-1)  
except:  
 pass  
## keep going (but x will not be set)

You can specify something you want to do with only a particular set of errors:

try:  
 x = math.sqrt(-1)  
except ValueError:   
 print("a ValueError occurred")  
except:  
 print("some other error occurred")  
## keep going (but x will not be set)

## a ValueError occurred

If the error isn’t caught because it isn’t the right type, it will act like it normally does (without the try:)

try:  
 z += 5 ## not defined yet  
except ValueError:   
 print("a ValueError occurred")

NameError: name 'z' is not defined

We could catch this with a general-purpose except:

try:  
 z += 5 ## not defined yet  
except ValueError:   
 print("a ValueError occurred")  
except:  
 print("some other error occurred")

## some other error occurred

Or add another clause to catch it:

try:  
 z += 5 ## not defined yet  
except ValueError:   
 print("a ValueError occurred")  
except NameError:  
 print("a NameError occurred")  
except:  
 print("some other error occurred")

## a NameError occurred

## general rules

* see if you can change your code to avoid getting errors in the first place
* catch specific errors
* do something sensible with errors (e.g. convert to warnings, return nan …)

try:  
 x = math.sqrt(-1)  
except ValueError:   
 x = math.nan  
print(x)

## nan

# pandas

## definition and reference

* pandas stands for **pan**el **da**ta **s**ystem. It’s a convenient and powerful system for handling large, complicated data sets. (The author [pronounces it “pan-duss”](https://twitter.com/wesmckinn/status/706661972431892483?lang=en).)
* [pandas cheat sheet](https://github.com/pandas-dev/pandas/blob/master/doc/cheatsheet/Pandas_Cheat_Sheet.pdf)

## Data frames

* rectangular data structure, looks a lot like an array.
* each column is a **Series**; each column can be of a different type
* rows and columns act differently
* can index by (column) labels as well as positions
* handles **missing data** (NaN)
* convenient plotting
* fast operations with keys
* lots of facilities for input/output

import pandas as pd ## standard abbreviation  
# The initial set of baby names and birth rates  
names = ['Bob','Jessica','Mary','John','Mel']  
births = [968, 155, 77, 578, 973]  
## initialize DataFrame with a \*dictionary\*  
p = pd.DataFrame({'Name': names, 'Count': births})  
print(p)

## Name Count  
## 0 Bob 968  
## 1 Jessica 155  
## 2 Mary 77  
## 3 John 578  
## 4 Mel 973

What can we do with it?

* “Simple” indexing
  + *Indexing* (a single value) selects a column by its *key*
  + key could be a number, if column names weren’t given when setting up the data frame
  + *Slicing* selects *rows* by number
  + indexing with a *list* gives multiple columns
  + .iloc gives row/column indices (like an array)

p["Count"] ## extract a column = Series (by \*name\*)  
p[2:3] ## slice one row (3-2 = 1)  
p[2:5] ## slice multiple rows  
p[["Name","Count"]] ## extract multiple columns (data frame)  
p.iloc[1,1] ## index with row/column integers like an array  
p.iloc[0:5,:] ## can also slice

Indexing by name

p["Name"][4] ## 5th element of Name  
p.Name ## attribute!  
p.loc[1:2,"Name"] ## index by \*label\*, \_inclusive\_

## Measles data

Download US measles data from [Project Tycho](https://www.tycho.pitt.edu/index.php).

* read\_csv reads a CSV file as a **data frame**; it automatically interprets the first row as headings
* df.iloc[] indexes the result as though it were an array
* df.head() shows just at the beginning; df.tail() shows just the end

Let’s look at the first few rows of a data set on measles in US states:

## "Weekly Measles Cases, 1909-2001"  
## ...  
## "Data provided by Project Tycho, Data Version 1.0.0, released 28 Novem...  
## "YEAR","WEEK","ALABAMA","ALASKA","AMERICAN SAMOA","ARIZONA","ARKANSAS"...  
## 1909,1,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-...  
## 1909,2,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-...  
## 1909,3,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-...

fn = "../data/MEASLES\_Cases\_1909-2001\_20150322001618.csv"  
p = pd.read\_csv(fn,skiprows=2,na\_values=["-"]) ## read in data  
p.head() ## look at the first little bit

## YEAR WEEK ALABAMA ALASKA ... WEST VIRGINIA WISCONSIN WYOMING Unnamed: 61  
## 0 1909 1 NaN NaN ... NaN NaN NaN NaN  
## 1 1909 2 NaN NaN ... NaN NaN NaN NaN  
## 2 1909 3 NaN NaN ... NaN NaN NaN NaN  
## 3 1909 4 NaN NaN ... NaN NaN NaN NaN  
## 4 1909 5 NaN NaN ... NaN NaN NaN NaN  
##   
## [5 rows x 62 columns]

Mostly NaN values at the beginning! (NaN = “not a number”: similar to nan from math or numpy)

## Selecting

* Like numpy array indexing, but a little different …
* Pandas doc, [indexing and selecting](http://pandas.pydata.org/pandas-docs/dev/indexing.html)
  + extract by name: df.loc[:,"MASSACHUSETTS":"NEVADA"] (index by *label*; **includes endpoint**)
  + extract by integer index: iloc method, df.iloc[:,range] (index by *integer*; **doesn’t include endpoint**)

p.loc[:,"MASSACHUSETTS":"NEVADA"]

## MASSACHUSETTS MICHIGAN MINNESOTA ... MONTANA NEBRASKA NEVADA  
## 0 NaN NaN NaN ... NaN NaN NaN  
## 1 NaN NaN NaN ... NaN NaN NaN  
## 2 NaN NaN NaN ... NaN NaN NaN  
## 3 NaN NaN NaN ... NaN NaN NaN  
## 4 NaN NaN NaN ... NaN NaN NaN  
## ... ... ... ... ... ... ... ...  
## 4856 NaN NaN NaN ... NaN NaN NaN  
## 4857 NaN NaN NaN ... NaN NaN NaN  
## 4858 NaN NaN NaN ... NaN NaN NaN  
## 4859 NaN NaN NaN ... NaN NaN NaN  
## 4860 NaN NaN NaN ... NaN NaN NaN  
##   
## [4861 rows x 8 columns]

This is the same:

pc = list(p.columns) ## list of colum names  
print(pc[:5])  
## find the locations of these two state names

## ['YEAR', 'WEEK', 'ALABAMA', 'ALASKA', 'AMERICAN SAMOA']

mass\_ind = list(pc).index("MASSACHUSETTS")  
neva\_ind = list(pc).index("NEVADA")  
## index using `.iloc` (with extended range)  
p.iloc[:,mass\_ind:neva\_ind+1]

## MASSACHUSETTS MICHIGAN MINNESOTA ... MONTANA NEBRASKA NEVADA  
## 0 NaN NaN NaN ... NaN NaN NaN  
## 1 NaN NaN NaN ... NaN NaN NaN  
## 2 NaN NaN NaN ... NaN NaN NaN  
## 3 NaN NaN NaN ... NaN NaN NaN  
## 4 NaN NaN NaN ... NaN NaN NaN  
## ... ... ... ... ... ... ... ...  
## 4856 NaN NaN NaN ... NaN NaN NaN  
## 4857 NaN NaN NaN ... NaN NaN NaN  
## 4858 NaN NaN NaN ... NaN NaN NaN  
## 4859 NaN NaN NaN ... NaN NaN NaN  
## 4860 NaN NaN NaN ... NaN NaN NaN  
##   
## [4861 rows x 8 columns]

## More examples

You can also refer to *individual* columns as **attributes** (i.e. just p.<name>)

p.ARIZONA[:5]

## 0 NaN  
## 1 NaN  
## 2 NaN  
## 3 NaN  
## 4 NaN  
## Name: ARIZONA, dtype: float64

p.ARIZONA.head()

## 0 NaN  
## 1 NaN  
## 2 NaN  
## 3 NaN  
## 4 NaN  
## Name: ARIZONA, dtype: float64

.drop() gets rid of elements

pp = p.drop(["YEAR","WEEK"],axis=1)  
## equivalent to  
pp2 = p.iloc[2:,]  
pp3 = p.loc[:,"ARIZONA"]

Always use name-indexing whenever you can!

.index is a special attribute of data frames that governs searching, plotting, etc.. Here we’ll set it to a decimal date value:

pp.index = p.YEAR+(p.WEEK-1)/52

## Filtering

Choosing specific rows of a data frame; &, | ,~ correspond to and, or, not (individual elements *must* be in parentheses)

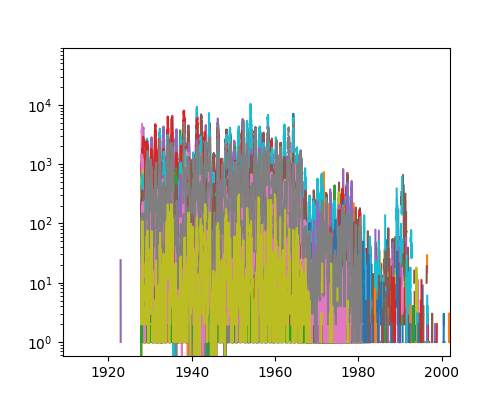
ariz = p.ARIZONA ## pull out a column (attribute)  
ariz[(p.YEAR==1970) & (ariz>50)] ## \*must\* use parentheses!

## 3196 69.0  
## 3197 57.0  
## 3198 62.0  
## 3200 56.0  
## 3203 73.0  
## 3205 54.0  
## 3209 55.0  
## Name: ARIZONA, dtype: float64

## Basic plotting

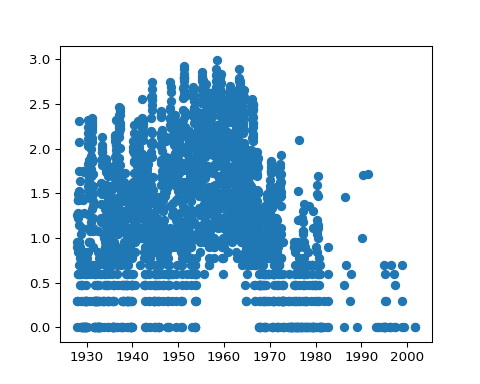
pandas will automatically plot data frames in a (reasonably) sensible way

import matplotlib.pyplot as plt  
fig, ax = plt.subplots()  
## pp.plot()  
pp.plot(legend=False,logy=True) ## plot method (non-Pythonic)  
plt.savefig("pix/measles1.png")



Or we can create our own (less complex) plots

import numpy as np  
fig = plt.figure()  
ax = fig.add\_subplot(1,1,1)  
ax.scatter(pp.index,np.log10(pp.ARIZONA))

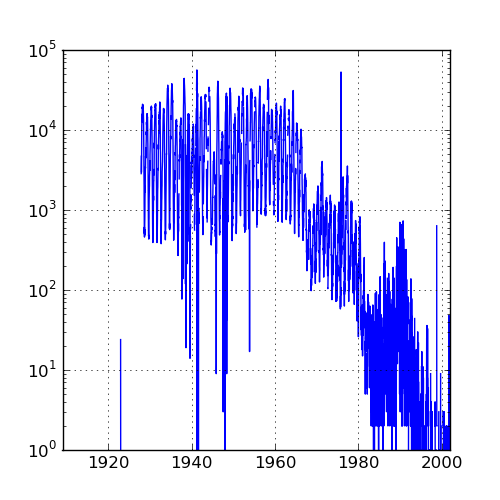


## Column and row manipulations

* totals by week

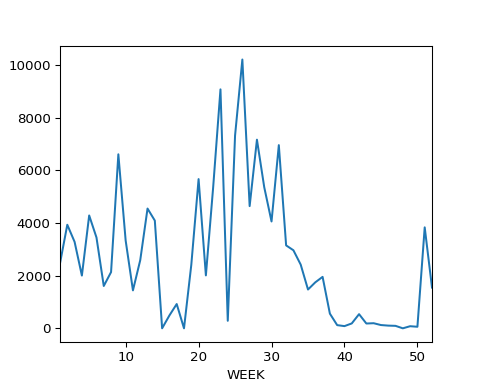
ptot = pp.sum(axis=1)

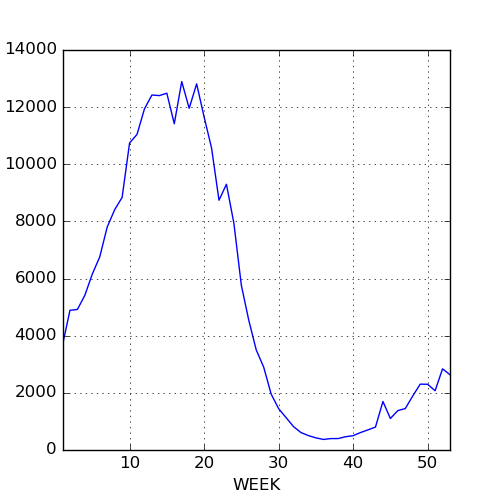
* df.min, df.max, df.mean all work too …



## Aggregation

ptotweek = ptot.groupby(p.WEEK)  
ptotweekmean = ptotweek.aggregate(np.mean)  
ptotweekmean.plot()





## Dates and times

[reference](https://docs.python.org/3.0/library/datetime.html)

* (Another) complex subject.
* Lots of [possible date formats](https://xkcd.com/1179/)
* Basic idea: something like %Y-%m-%d; separators just match whatever’s in your data (usually “/” or “-”). Results need to be unambiguous, and ambiguity is dangerous (how is day of month specified? lower case, capital? etc.)
* pandas tries to guess, but you shouldn’t let it.

print(pd.to\_datetime("05-01-2004"))

## 2004-05-01 00:00:00

print(pd.to\_datetime("05-01-2004",format="%m-%d-%Y"))

## 2004-05-01 00:00:00

* Time zones and daylight savings time can be a nightmare
* May need to have the right number of digits, especially in the absence of separators:

import pandas as pd  
print(pd.to\_datetime("1212004",format="%m%d%Y"))

## 2004-12-01 00:00:00

print(pd.to\_datetime("12012004",format="%m%d%Y"))

## 2004-12-01 00:00:00

For our measles data we have week of year, so things get a little complicated

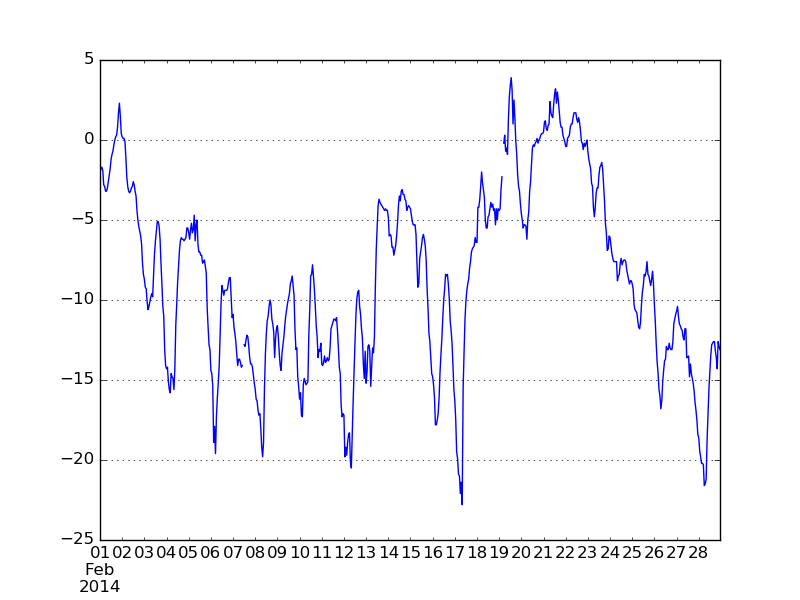
yearstr = p.YEAR.apply(format)  
weekstr = p.WEEK.apply(format,args=["02"])  
datestr = p.YEAR.astype(str)+"-"+weekstr+"-0"  
dateindex = pd.to\_datetime(datestr,format="%Y-%U-%w")

## Binning results

* turn a quantitative variable into categories
* pd.cut(x,bins=...); decide on bins
* pd.qcut(x,n); decide on number of bins (equal occupancy)

## Weather data

## fancy stuff: automatically look for index and convert it to a date/time  
p = pd.read\_csv("../data/eng2.csv",skiprows=14,encoding="latin1",index\_col="Date/Time",parse\_dates=True)  
## rename columns  
p.columns = [  
 'Year', 'Month', 'Day', 'Time', 'Data Quality', 'Temp (C)',   
 'Temp Flag', 'Dew Point Temp (C)', 'Dew Point Temp Flag',   
 'Rel Hum (%)', 'Rel Hum Flag', 'Wind Dir (10s deg)', 'Wind Dir Flag',   
 'Wind Spd (km/h)', 'Wind Spd Flag', 'Visibility (km)', 'Visibility Flag',  
 'Stn Press (kPa)', 'Stn Press Flag', 'Hmdx', 'Hmdx Flag', 'Wind Chill',   
 'Wind Chill Flag', 'Weather']  
## drop columns that are \*all\* NA  
p = p.dropna(axis=1,how='all')  
p["Temp (C)"].plot()  
## get rid of columns (axis=1) we don't want  
p = p.drop(['Year', 'Month', 'Day', 'Time', 'Data Quality'], axis=1)



Now pull out the temperature and take the median by hour:

temp = p[['Temp (C)']]  
temp["Hour"] = temp.index.hour

## <string>:1: SettingWithCopyWarning:   
## A value is trying to be set on a copy of a slice from a DataFrame.  
## Try using .loc[row\_indexer,col\_indexer] = value instead  
##   
## See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

temphr = temp.groupby('Hour')  
medtmp = temphr.aggregate(np.median)  
maxtmp = temphr.aggregate(np.max)  
mintmp = temphr.aggregate(np.min)

Now plot these …

