STAT 177, CLASS 5

Richard Waterman

July 2020

OBJECTIVES

OBJECTIVES

- Writing basic functions.
- Common data cleaning activities.
- More on the “groupby” command.
- From transactional to behavioral data sets.

WRITING BASIC FUNCTIONS

WRITING BASIC FUNCTIONS

- If you find yourself using almost identical blocks of code in different parts of your program, then you almost certainly want to write a “function”.
- A function helps you organize and reuse code.
- In math, a function f is a rule that takes an input x and provides a unique output, y .
- We write: $y = f(x)$.
- When we write Python code for a function, we have to define the inputs and create the rule that makes the output.
- Functions will be useful to us when we do data analysis, because we are often repeating the same activity on different data sets, or even the same activity within a dataset.
- These activities, could be data cleaning, merging, summarizing and so on.
- We will learn how to write a function, then later see how they can be applied to data.
- There is a special type of function in Python called a “lambda function” that is useful to create very simple functions.

COMPONENTS OF A FUNCTION

- The Python keyword to create a function is ‘def’.
- The keyword to end a function and return its result is ‘return’.
- The body of a function needs to be indented.
- The function can take multiple inputs (parameters/arguments).
- These inputs can be specified as either positional inputs or keyword inputs.
- The keyword inputs are typically used for those arguments that take default values or are optional.
- We will write some functions to illustrate the structure.
- The function is *called* (run) by using its name and then parentheses().

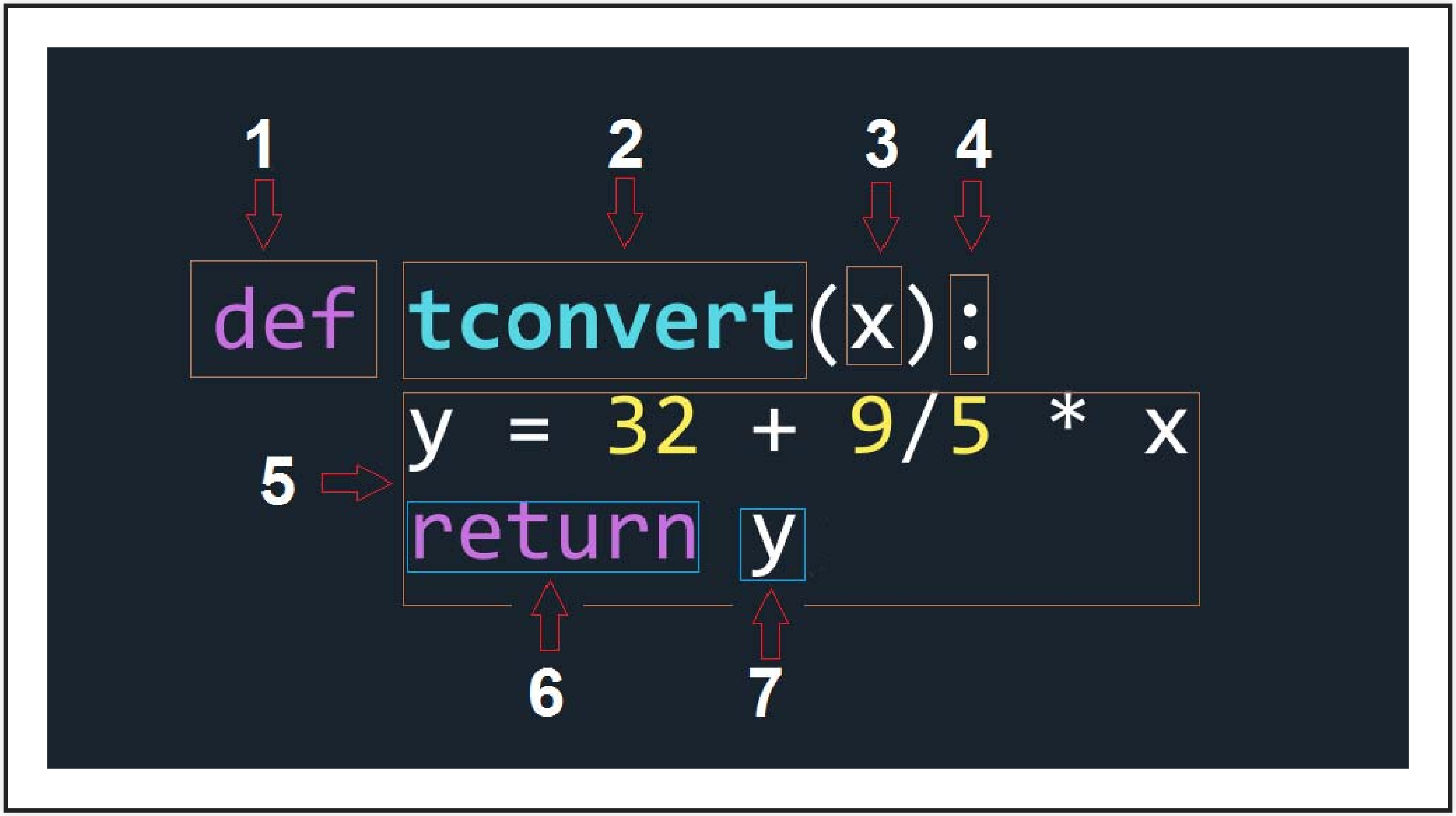
DEGREES CELSIUS TO FAHRENHEIT

- We will write a function, called ‘tconvert’, that takes a single input and converts Celsius to Fahrenheit.
- The conversion formula is $f(c) = 32 + (9/5)c$, where c is the input temperature.

```
def tconvert(x):
    y = 32 + 9/5 * x
    return y

# Call the function a couple of times
print(tconvert(22))
print(tconvert(100))
```

```
71.6
212.0
```

ANATOMY OF THE FUNCTION



```
def tconvert(x):
    y = 32 + 9/5 * x
    return y
```

ANATOMY OF THE FUNCTION (CTD.)

1. The keyword “def” signals a function is about to be defined.
2. This particular function is named *tconvert* .
3. The *parameter* to be input is called “x”.
4. The colon indicates we are about to get an indented code block.
5. The *body* of the function.
6. The keyword “return” statement.
7. The value to be returned by the function.

ADDING A DOCSTRING TO THE FUNCTION

- The docstring describes the action of the function.
- If it is there, then it is the first string right after the function definition.
- Using three quotes, ’’’, allows you the potential to create a multiline string.
- You can print the docstring using the ‘.__doc__’ method as below.

```
def tconvert(x):
    '''Takes in a number x, returns the converted temperature of x.'''
    y = 32 + 9/5 * x
    return y

print(tconvert.__doc__)
```

```
Takes in a number x, returns the converted temperature of x.
```

CHECKING THE INPUT TYPE

- Look what happens if we call this function on a string input:

```
print(tconvert("It's hot!"))
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-3-bbb8240b6a68> in <module>
----> 1 print(tconvert("It's hot!"))


<ipython-input-2-fcc95a68b6ad> in tconvert(x)
      1 def tconvert(x):
      2     '''Takes in a number x, returns the converted temperature of x.'''
----> 3     y = 32 + 9/5 * x
      4     return y
      5
```

CHECKING THE INPUT TYPE

- It makes sense to only have the function work if it has a numeric input.
- The input type can be checked by using the “isinstance” command.

```
def tconvert(x):
    '''Takes in a number x, returns the converted temperature of x.'''
    if( isinstance(x, (int, float)) and not isinstance(x, (bool))):  #Booleans are also ints!
        y = 32 + 9/5 * x
        return y
    else:
        return "This function only works with numbers."

print(tconvert(3))
print(tconvert(21.2))
print(tconvert(True))
print(tconvert("It's hot!"))
```

```
37.4
70.16
This function only works with numbers.
This function only works with numbers.
```

MULTIPLE PARAMETERS

- We will give the function an additional argument, that allows the possibility of going from ‘f2c’ or ‘c2f’.
- We check that it takes on one of the two valid values.

```python
def tconvert(x, direction):
    '''Takes in a number x, returns the converted temperature of x.'''

    if not ( (direction == "f2c") or (direction == "c2f")):
        return "Direction must be 'c2f' or 'f2c'."
    if not ( isinstance(x, (int, float)) and not isinstance(x, (bool))): # Booleans are also ints!
        return "This function only works with numbers."

    if direction == "c2f":
        y = 32 + 9/5 * x
        return y
    if direction == "f2c":
        y = (x - 32) * 5/9
        return y
```

USING THE IMPROVED FUNCTION

```
print(tconvert(0.0, "c2f"))
print(tconvert(71, "f2c"))
print(tconvert(71, "a2b")) # Break it on purpose.
print(tconvert(225,"c2f"))
```

```
32.0
21.666666666666668
Direction must be 'c2f' or 'f2c'.
437.0
```

KEYWORD ARGUMENTS

- You can also provide arguments with the key = value syntax.
- This way, if there are multiple arguments, the order of the arguments does not matter.

```
print(tconvert(direction = "f2c", x = 21))
```

```
-6.111111111111111
```

KEYWORD ARGUMENTS CAN’T COME BEFORE POSITIONAL ARGUMENTS

- There are rules for the order of the arguments.

```
print(tconvert(direction = "f2c", 212)) # This is illegal.
```

```
  File "<ipython-input-8-5f11852d7b81>", line 1
    print(tconvert(direction = "f2c", 212)) # This is illegal.
                                     ^
SyntaxError: positional argument follows keyword argument
```

DEFAULT VALUES

- When a function has many arguments it makes sense to give the less used ones, default values.
- Using the ‘parameter = value’ syntax will do this.
- We will assume that the user wants to go from c2f, unless they say otherwise.
- Note the ‘direction = “c2f”’ in the argument list below.

```
def tconvert(x, direction = "c2f"):
    '''Takes in a number x, returns the converted temperature of x.'''

    if not ( (direction == "f2c") or (direction == "c2f")):
        return "Direction must be 'c2f' or 'f2c'."
    if not ( isinstance(x, (int, float)) and not isinstance(x, (bool))): # Booleans are also ints!
        return "This function only works with numbers."

    if direction == "c2f":
        y = 32 + 9/5 * x
        return y
    if direction == "f2c":
        y = (x - 32) * 5/9
        return y
```

CALLING THE FUNCTION WITH THE DEFAULT PARAMETER

```
print(tconvert(10)) # No need for the direction parameter, if happy with 'c2f'
print(tconvert(200))
```

```
50.0
392.0
```

RETURNING MORE THAN ONE VALUE

- You can return multiple values from a function as a ‘tuple’.
- The function below takes a pandas series and returns the top 3 occurring levels.
- We make some data by randomly sampling letters of the alphabet.

```
def top3(x):
    y = x.value_counts(sort=True) # Recall thatvalue_counts makes frequencies, and the sort argument
  will sort them.
    return y.index[0], y[0], y.index[1], y[1], y.index[2], y[2] # Return multiple values.
```

```
import pandas as pd
import numpy as np
np.random.seed(1234)
data = np.random.choice(['a','b','c','d','e','f','g','h','i','j'], size=1000, replace=True) # Random
  sampling with replacement.
data = pd.Series(data) # Store the data in a pandas Series.
```

RETURNING MULTIPLE VALUES

- Note how the results come back as a tuple.

```
top3(data)
```

```
('j', 116, 'd', 108, 'b', 107)
```

LAMBDA FUNCTIONS

- These are special functions, with no name (anonymous), that are defined in an expression rather than a statement.
- They are suitable for very simple functions that are used for a short period of time.
- They are useful in data analysis, because we are often trying different transforms of the data, and we can make a function, which itself takes another function (the lambda function) as an argument.

```python
import math
def data_transform(x, fn):
    return [fn(input) for input in x] # Notice that the function "fn" is being used in the comprehension.

data = [1,2,5,6,2,1]

print(data_transform(data, lambda y: y**2))
print(data_transform(data, lambda y: math.log(y)))
```

```
[1, 4, 25, 36, 4, 1]
[0.0, 0.6931471805599453, 1.6094379124341003, 1.791759469228055, 0.6931471805599453, 0.0]
```

THE ‘MAP’ COMMAND

- Another place you might see lambda functions used is with the “map” command.
- ‘map’ will map a function to the elements of a container.
- You may see ‘map’ in someone else’s Python code, but many prefer a list comprehension instead.

```
data = np.random.randint(10, size=10000) # Make some data, random integers between 0 and 9.
```

```
list(map(lambda x: x**2 , data))[0:5] # Just look at the first 5 elements.
```

```
[25, 49, 0, 4, 9]
```

MORE ABOUT FUNCTIONS

- There’s more to learn about functions, such as ‘Arbitrary Arguments’ (*args) and ‘Arbitrary Keyword Arguments’ (**kwargs), but we will discuss these as they become necessary.

COMMON DATA CLEANING ACTIVITIES

COMMON DATA CLEANING ACTIVITIES

- Truth be told, what is ***common*** most likley depends on the type of data you are used to.
- Cleaning a time-series could be quite different from cleaning a list of words.
- Examples of cleaning/pre-processing activities include:
    - There are rows in the data frame that need to be removed.
    - There is missing data that needs to be addressed.
    - There are typos in the data that need to be corrected.
    - Things that should be numeric are held as strings.
    - Dates have not been properly parsed.
    - Levels of a categorical variable need to be collapsed.
    - Sorting data.

REMOVING ROWS FROM THE DATA FRAME

- We will read in a data frame and remove rows.
    - Remove the first row.
    - Remove the last row.
    - Remove rows using a logical filter.

```
# Read in a data frame for illustration.
import os
os.chdir('C:\\Users\\richardw\\Dropbox (Penn)\\Teaching\\477s2020\\DataSets')
op_data = pd.read_csv("Outpatient_to_clean.csv")
print(op_data.shape) # Track the dimensions.
op_data.drop(0, inplace=True) # Drop the first row.
print(op_data.shape) # Track the dimensions.
op_data.drop(len(op_data)-1, inplace=True) # Drop the last row.
print(op_data.shape) # Track the dimensions.
```

```
(3699, 9)
(3698, 9)
(3697, 9)
```

REMOVE ROWS USING A LOGICAL FILTER

```
temp = op_data.loc[op_data['Status'] != "Bumped"] # The logical filter selects all rows, that are not Bumped.
temp['Status'].value_counts() # Note that "Bumped has gone".
```

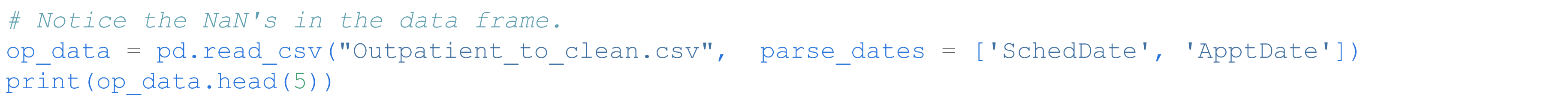
```
Arrived        2163
Cancelled       796
No Show         526
Rescheduled       1
Name: Status, dtype: int64
```

USING THE READ_CSV ARGUMENTS

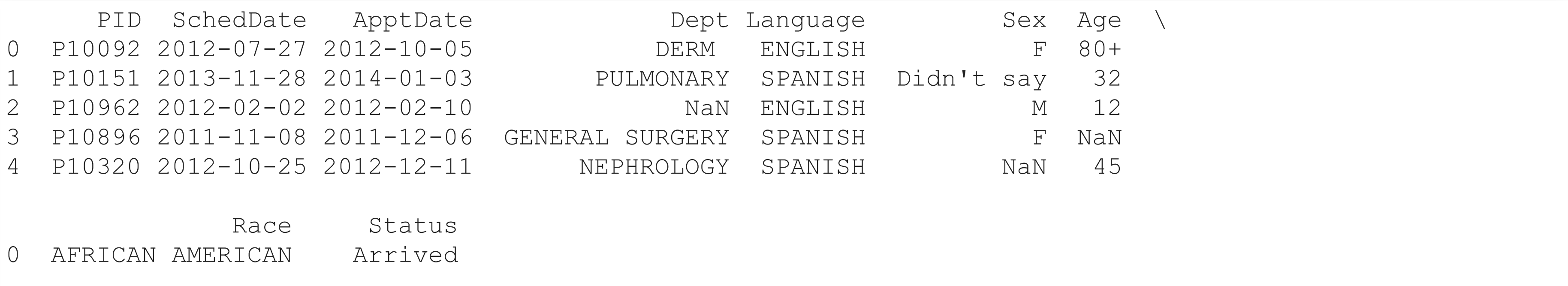
- If you know upfront that that certain rows at the beginning or end of the data frame need to be dropped, you could use the arguments to read_csv:
- You can specify the specific line numbers to skip, or number of lines at the bottom of the file to skip with the arguments:
  - skiprows
  - skipfooter

MISSING DATA

- It's very common to have missing data in most analyses.
- The most used approach is to remove rows with missing data (called "case-wise" deletion).
- You may also "fill in" (impute) the missing data.
- If it is a categorical variable, you could potentially define a new category "Missing" for the missing data.
- In pandas the convention to indicate missing data is NA (just like R uses).



```
# Notice the NaN's in the data frame.
op_data = pd.read_csv("Outpatient_to_clean.csv",  parse_dates = ['SchedDate', 'ApptDate'])
print(op_data.head(5))
```



```
      PID  SchedDate   ApptDate             Dept Language         Sex  Age  \
0  P10092 2012-07-27 2012-10-05             DERM  ENGLISH           F  80+
1  P10151 2013-11-28 2014-01-03        PULMONARY  SPANISH  Didn't say   32
2  P10962 2012-02-02 2012-02-10              NaN  ENGLISH           M   12
3  P10896 2011-11-08 2011-12-06  GENERAL SURGERY  SPANISH           F  NaN
4  P10320 2012-10-25 2012-12-11       NEPHROLOGY  SPANISH         NaN   45

               Race   Status
0  AFRICAN AMERICAN  Arrived
```

```
1          HISPANIC  Cancelled
2  AFRICAN AMERICAN        NaN
3          HISPANIC     Bumped
4          HISPANIC    No Show
```

USING .DROPNA()

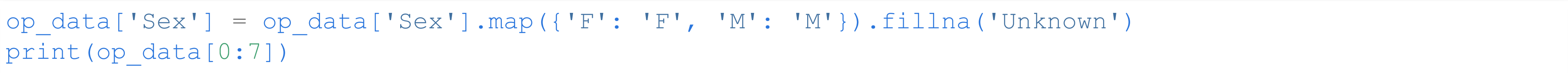
- The .dropna() would be a common pre-processing step.

```
op_data.dropna(inplace=True) # '.dropna()' removes all rows with any missing data.
print(op_data.head(5)) # Note that some of the rows have disappeared.
```

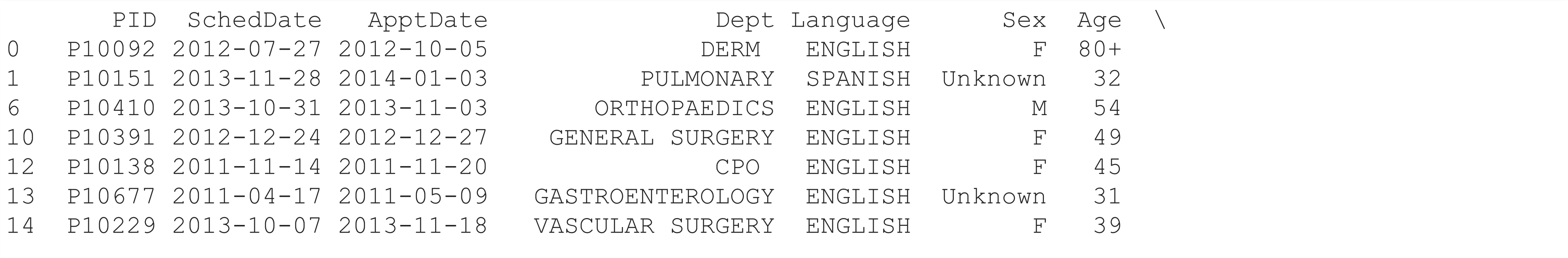
```
       PID  SchedDate   ApptDate             Dept Language          Sex  Age  \
0   P10092 2012-07-27 2012-10-05            DERM   ENGLISH           F  80+
1   P10151 2013-11-28 2014-01-03        PULMONARY  SPANISH  Didn't say   32
6   P10410 2013-10-31 2013-11-03     ORTHOPAEDICS  ENGLISH           M   54
10  P10391 2012-12-24 2012-12-27  GENERAL SURGERY  ENGLISH           F   49
12  P10138 2011-11-14 2011-11-20              CPO   ENGLISH           F   45

                    Race     Status
0       AFRICAN AMERICAN    Arrived
1               HISPANIC  Cancelled
6       AFRICAN AMERICAN    No Show
10  WHITE (NON-HISPANIC)    Arrived
12              HISPANIC    Arrived
```

TYPOS AND STRINGS

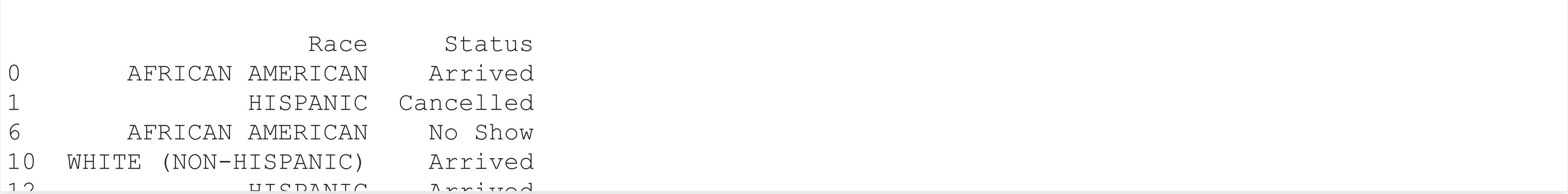
- If you have a typo, then there is the possibility to make it missing, or to overwrite the bad values if you know what they should be.
- In the outpatient data all values for Sex should be either ‘M’ or ‘F’
- We will look through the Sex column, changing elements that are neither ‘M’ or ‘F’ to ‘Unknown’.
- If the .map method doesn’t find a key in the dict structure, for the element, then it replaces it with NA.
- The .fillna, then replaces these NAs with ‘Unknown’.



```
op_data['Sex'] = op_data['Sex'].map({'F': 'F', 'M': 'M'}).fillna('Unknown')
print(op_data[0:7])
```



```
       PID  SchedDate   ApptDate               Dept Language      Sex  Age  \
0   P10092 2012-07-27 2012-10-05              DERM   ENGLISH        F  80+
1   P10151 2013-11-28 2014-01-03          PULMONARY  SPANISH  Unknown   32
6   P10410 2013-10-31 2013-11-03       ORTHOPAEDICS  ENGLISH        M   54
10  P10391 2012-12-24 2012-12-27    GENERAL SURGERY  ENGLISH        F   49
12  P10138 2011-11-14 2011-11-20               CPO   ENGLISH        F   45
13  P10677 2011-04-17 2011-05-09   GASTROENTEROLOGY  ENGLISH  Unknown   31
14  P10229 2013-10-07 2013-11-18   VASCULAR SURGERY  ENGLISH        F   39
```



```
                   Race     Status
0      AFRICAN AMERICAN    Arrived
1              HISPANIC  Cancelled
6      AFRICAN AMERICAN    No Show
10  WHITE (NON-HISPANIC)    Arrived
12             HISPANIC    Arrived
```

REMOVING WHITE SPACE

- In this data frame the elements of the ‘Dept’ column are inconsistent.
- Some of them start/end with blank spaces.
- We will remove all white space from the strings in ‘Dept’ using a method called .str.strip().

```
op_data['Dept'] = op_data['Dept'].str.strip()
print(op_data['Dept'])
```

```
0                   DERM
1              PULMONARY
6           ORTHOPAEDICS
10       GENERAL SURGERY
12                   CPO
              ...
3694        RHEUMATOLOGY
3695      OTOLARYNGOLOGY
3696    VASCULAR SURGERY
3697        NEUROLOGICAL
3698                DERM
Name: Dept, Length: 3688, dtype: object
```

DATA THAT SHOULD BE NUMERIC ARE HELD AS STRINGS

- In this dataset there is an Age value recorded as ‘80+’.
- This is enough for the columns to be read in as a string.
- The to_numeric function will convert the strings to numbers if possible, but if not, replace them with NA.

```
print(op_data[0:2])
op_data['Age'].mean() # This fails.
```

```
      PID  SchedDate   ApptDate       Dept Language      Sex  Age  \
0  P10092 2012-07-27 2012-10-05       DERM  ENGLISH        F  80+
1  P10151 2013-11-28 2014-01-03  PULMONARY  SPANISH  Unknown   32

               Race     Status
0  AFRICAN AMERICAN    Arrived
1          HISPANIC  Cancelled



---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)
```

CONVERTING FROM STRING TO NUMERIC

- The “errors = ‘coerce’” argument below will replace the non-coercible elements with NAs.

```
op_data['Age'] = pd.to_numeric(op_data['Age'], errors = 'coerce')
print(op_data['Age'].mean()) # Now we can calculate the mean.
```

```
45.64065592309867
```

DATES HAVE NOT BEEN PROPERLY PARSED

- As we saw in the class 4 notes, there are formatting methods, that you can use to control exactly how to parse a string as a date.
- See Class 4 notes, ‘pd.to_datetime’, slides 7.5 and 7.7.

LEVELS OF A CATEGORICAL VARIABLE NEED TO BE COLLAPSED

- This is a very common task and there will be many ways to do it.
- One that we have already seen is to use the .map method.
- But if there are many levels this could be cumbersome.
- In this data set the are two ‘Dept’ levels, NEUROLOGICAL and NEUROLOGY, that should be the same level.
- We can search for one, and replace it with the other:

THE ORIGINAL FREQUENCY DISTRIBUTION

```
print(op_data['Dept'].value_counts())
```

```
NEUROLOGICAL                 668
CPO                          657
ORTHOPAEDICS                 381
DERM                         236
PLASTIC SURGERY              221
GENERAL SURGERY              179
UROLOGY                      177
GASTROENTEROLOGY             162
PODIATRY                     161
VASCULAR SURGERY             124
NEPHROLOGY                   122
RHEUMATOLOGY                 116
OTOLARYNGOLOGY               114
PULMONARY                    108
TRAUMA                        89
```

REPLACING A SINGLE LEVEL

- We use the '.loc' method, creating a logical filter for the rows we want to change.

```
op_data.loc[op_data['Dept'] == "NEUROLOGICAL", 'Dept'] = "NEUROLOGY"  # The logical replacement filter.
op_data['Dept'].value_counts() # Now there are more rows in the NEUROLOGY column.
```

```
NEUROLOGY                     711
CPO                           657
ORTHOPAEDICS                  381
DERM                          236
PLASTIC SURGERY               221
GENERAL SURGERY               179
UROLOGY                       177
GASTROENTEROLOGY              162
PODIATRY                      161
VASCULAR SURGERY              124
NEPHROLOGY                    122
RHEUMATOLOGY                  116
OTOLARYNGOLOGY                114
PULMONARY                     108
TRAUMA                         89
```

SORTING A DATA FRAME

- You can sort by the Index (rownames) or by the values in a column.
- You can also sort on multiple columns.

```
op_data.sort_values(by='Age', inplace = True)
print(op_data.head(3))
```

```
         PID  SchedDate   ApptDate               Dept Language Sex   Age  \
3294  P10453 2013-05-22 2013-05-22  PEDIATRIC SURGERY  ENGLISH   M  0.0
821   P10453 2013-05-20 2013-05-22  PEDIATRIC SURGERY  ENGLISH   M  0.0
2546  P10993 2014-01-14 2014-01-20  PEDIATRIC SURGERY  ENGLISH   M  0.0

                  Race     Status
3294  AFRICAN AMERICAN    Arrived
821   AFRICAN AMERICAN  Cancelled
2546  AFRICAN AMERICAN    Arrived
```

SORT BY PATIENT PID, THEN SCHEDULE DATE WITHIN PID

```
op_data.sort_values(by = ['PID', 'SchedDate'], inplace=True)  # Sorting by two columns.
print(op_data.head(5))
```

```
         PID  SchedDate   ApptDate             Dept Language Sex   Age  \
2598  P10001 2012-11-23 2012-11-26        NEUROLOGY  ENGLISH   M  45.0   
499   P10002 2011-12-02 2011-12-21        NEUROLOGY  ENGLISH   M  11.0   
2733  P10002 2011-12-21 2012-06-20        NEUROLOGY  ENGLISH   M  11.0   
1618  P10003 2013-10-18 2014-02-16  PLASTIC SURGERY  ENGLISH   M  11.0   
3043  P10004 2012-09-26 2012-10-08        NEUROLOGY  BENGALI   M  39.0   

                      Race     Status  
2598  WHITE (NON-HISPANIC)    Arrived  
499   WHITE (NON-HISPANIC)    Arrived  
2733  WHITE (NON-HISPANIC)  Cancelled  
1618      AFRICAN AMERICAN    Arrived  
3043                 OTHER    Arrived  
```

THE GROUPBY COMMAND

THE GROUPBY COMMAND

- Summarizing a data set by the levels of a categorical variable is an essential activity.
- We saw ‘groupby’ used with .value_counts() to summarize over the day-of-week or month-of-year.
- But it is very flexible and can be used to summarize in a more sophisticated way.
- In our outpatient dataset we may wish to summarize by patient ID (PID) to create a behavioral view of the patient.
- We will start by counting the number of appointments each patient has had.

TRANSACTIONS TO BEHAVIOR

- Many databases are designed to capture *transactions* .
- But many analytic techniques are applied to behavioral patterns.
- This requires individual transactions to be aggregated to behaviors via a common ID.

EXAMPLES

- Hospital:
  - The patient admissions database shows you who is in the hospital.
  - Combining admissions gives you a patient’s history.
- Supermarket:
  - Individual items get aggregated to a customer’s shopping visit.
  - Customer visits get aggregated to a long term behavior.
- HR:
  - A cut of the employee database in any given month, shows you who is employed.
  - Combining the months, shows you an employee’s history.
- Outpatient clinic:
  - The visits database tells you about each visit.
  - Combining the visits, shows you a patient’s history.

THE COMMONALITY

- All of these examples are of the same essential nature.
- Transactions are recorded.
- A common ID can be used to link transactions.
- Histories can be built across common IDs.
- This translates transactions to {} behavior/history.

COUNTING PATIENT VISITS

- groupby and using the count method will tell us how many appointments each patient had.

```
op_data.groupby('PID')['PID'].count()
```

```
PID
P10001     1
P10002     2
P10003     1
P10004     4
P10005     1
          ..
P10996     4
P10997    10
P10998    29
P10999     2
P11000     5
Name: PID, Length: 996, dtype: int64
```

PRIOR VISITS

- If we are to use the behavioral history as a predictor, we need to use prior visits, not including the latest one.

```
op_data.groupby('PID')['PID'].count() - 1 # Remove 1 from the count.
```

```
PID
P10001     0
P10002     1
P10003     0
P10004     3
P10005     0
          ..
P10996     3
P10997     9
P10998    28
P10999     1
P11000     4
Name: PID, Length: 996, dtype: int64
```

GROUPBY AND AGGREGATE

- We can summarize more than one variable at a time.
- The .agg method lets you create a new set of columns, based on a function applied to the groups.
- You can define this function yourself.
- Below we count the number of prior visits, and the number of prior visits that were of type “Arrived”.

```
prev = lambda x: len (x) - 1 # A lambda function to count previous appointments.
prior_arrived = lambda x: sum(x[:-1] == "Arrived") if len(x) > 1 else 'NA'  # Counting the number of
  prior arrivals.

op_behave = op_data.groupby('PID').agg(PriorVisits=('PID', prev),
                                       PriorArrived=('Status', prior_arrived))
print(op_behave.head(3))
```

```
        PriorVisits PriorArrived
PID
P10001            0           NA
P10002            1            1
P10003            0           NA
```

KEEPING THE FEATURES OF THE MOST RECENT VISIT

- We still want to make sure we have the features of the current visit too.
- The '.last' method will pull off the 'last' (most recent) visit.

```
op_last_values = op_data.groupby('PID').last() # Obtain the last row for each patient
print(op_last_values.head(5))
```

```
        SchedDate   ApptDate             Dept Language Sex   Age  \
PID
P10001 2012-11-23 2012-11-26        NEUROLOGY  ENGLISH   M  45.0
P10002 2011-12-21 2012-06-20        NEUROLOGY  ENGLISH   M  11.0
P10003 2013-10-18 2014-02-16  PLASTIC SURGERY  ENGLISH   M  11.0
P10004 2013-05-19 2013-05-26         PODIATRY  BENGALI   M  40.0
P10005 2013-06-07 2013-07-29             DERM  ENGLISH   F  24.0

                        Race     Status
PID
P10001  WHITE (NON-HISPANIC)    Arrived
P10002  WHITE (NON-HISPANIC)  Cancelled
P10003      AFRICAN AMERICAN    Arrived
P10004                 OTHER    Arrived
P10005  WHITE (NON-HISPANIC)    No Show
```

CHECKING THE CODE

- It is always a good idea to check your work.
- Here we pull off one patient and can then compare to the original sorted .csv file, to make sure it seems right.

```
op_last_values.loc['P10998']
```

```
SchedDate     2013-02-22 00:00:00
ApptDate      2013-03-02 00:00:00
Dept                          CPO
Language                  ENGLISH
Sex                             M
Age                            60
Race             AFRICAN AMERICAN
Status                    Arrived
Name: P10998, dtype: object
```

MERGE THE TWO DATA FRAMES

- We finish by merging the two new data frames on the PID, to create a single data set for analysis.

```
final_op_data = pd.merge(op_last_values, op_behave, on = 'PID')
print(final_op_data.head(5))
```

```
        SchedDate   ApptDate             Dept Language Sex   Age  \
PID
P10001 2012-11-23 2012-11-26        NEUROLOGY  ENGLISH   M  45.0
P10002 2011-12-21 2012-06-20        NEUROLOGY  ENGLISH   M  11.0
P10003 2013-10-18 2014-02-16  PLASTIC SURGERY  ENGLISH   M  11.0
P10004 2013-05-19 2013-05-26         PODIATRY  BENGALI   M  40.0
P10005 2013-06-07 2013-07-29             DERM  ENGLISH   F  24.0

                        Race     Status  PriorVisits PriorArrived
PID
P10001  WHITE (NON-HISPANIC)    Arrived            0           NA
P10002  WHITE (NON-HISPANIC)  Cancelled            1            1
P10003      AFRICAN AMERICAN    Arrived            0           NA
P10004                 OTHER    Arrived            3            2
P10005  WHITE (NON-HISPANIC)    No Show            0           NA
```

SUMMARY

SUMMARY

- Writing basic functions.
- Common data cleaning activities.
- More on the “groupby” command.
- From transactional to behavioral data sets.

NEXT TIME

NEXT TIME

- Graphics