❮ (../02-index-slice-subset/)

Python for ecologists (../)

Data Types and Formats

❯ (../04-merging-data/)

Overview

Teaching: 20 min
Exercises: 25 min
Questions

- What types of data can be contained in a DataFrame?
- Why is the data type important?

Objectives

- Describe how information is stored in a Python DataFrame.
- Define the two main types of data in Python: text and numerics.
- Examine the structure of a DataFrame.
- Modify the format of values in a DataFrame.
- Describe how data types impact operations.
- Define, manipulate, and interconvert integers and floats in Python.
- Analyze datasets having missing/null values (NaN values).

The format of individual columns and rows will impact analysis performed on a dataset read into python. For example, you can't perform mathematical calculations on a string (text formatted data). This might seem obvious, however sometimes numeric values are read into python as strings. In this situation, when you then try to perform calculations on the string-formatted numeric data, you get an error.

In this lesson we will review ways to explore and better understand the structure and format of our data.

Types of Data

How information is stored in a DataFrame or a python object affects what we can do with it and the outputs of calculations as well. There are two main types of data that we're explore in this lesson: numeric and text data types.

Numeric Data Types

Numeric data types include integers and floats. A **floating point** (known as a float) number has decimal points even if that decimal point value is 0. For example: 1.13, 2.0 1234.345. If we have a column that contains both integers and floating point numbers, Pandas will assign the entire column to the float data type so the decimal points are not lost.

An **integer** will never have a decimal point. Thus if we wanted to store 1.13 as an integer it would be stored as 1. Similarly, 1234.345 would be stored as 1234. You will often see the data type `Int64` in python which stands for 64 bit integer. The 64 simply refers to the memory allocated to store data in each cell which effectively relates to how many digits it can store in each "cell". Allocating space ahead of time allows computers to optimize storage and processing efficiency.

Text Data Type

Text data type is known as Strings in Python, or Objects in Pandas. Strings can contain numbers and / or characters. For example, a string might be a word, a sentence, or several sentences. A Pandas object might also be a plot name like ‘plot1’. A string can also contain or consist of numbers. For instance, ‘1234’ could be stored as a string. As could ‘10.23’. However **strings that contain numbers can not be used for mathematical operations**!

Pandas and base Python use slightly different names for data types. More on this is in the table below:

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime (http://doc.python.org/2/library/datetime.html) module in Python’s standard library)	Values meant to hold time data. Look into these for time series experiments.

Checking the format of our data

Now that we’re armed with a basic understanding of numeric and text data types, let’s explore the format of our survey data. We’ll be working with the same `surveys.csv` dataset that we’ve used in previous lessons.

```
# note that pd.read_csv is used because we imported pandas as pd
surveys_df = pd.read_csv("data/surveys.csv")
```

Remember that we can check the type of an object like this:

```
type(surveys_df)
```

OUTPUT: `pandas.core.frame.DataFrame`

Next, let’s look at the structure of our surveys data. In pandas, we can check the type of one column in a DataFrame using the syntax `dataFrameName[column_name].dtype`:

```
surveys_df['sex'].dtype
```

OUTPUT: `dtype('O')`

A type ‘O’ just stands for “object” which in Pandas’ world is a string (text).

```
surveys_df['record_id'].dtype
```

OUTPUT: `dtype('int64')`

The type `int64` tells us that python is storing each value within this column as a 64 bit integer. We can use the `dat.dtypes` command to view the data type for each column in a DataFrame (all at once).

```
surveys_df.dtypes
```

which **returns**:

```
record_id            int64
month                int64
day                  int64
year                 int64
plot_id              int64
species_id          object
sex                 object
hindfoot_length    float64
weight             float64
dtype: object
```

Note that most of the columns in our Survey data are of type `int64`. This means that they are 64 bit integers. But the weight column is a floating point value which means it contains decimals. The `species_id` and `sex` columns are objects which means they contain strings.

Working With Integers and Floats

So we’ve learned that computers store numbers in one of two ways: as integers or as floating-point numbers (or floats). Integers are the numbers we usually count with. Floats have fractional parts (decimal places). Let’s next consider how the data type can impact mathematical operations on our data. Addition, subtraction, division and multiplication work on floats and integers as we’d expect.

```
print(5+5)
10

print(24-4)
20
```

If we divide one integer by another, we get a float. The result on python 3 is different than in python 2, where the result is an integer (integer division).

```
print(5/9)
0.5555555555555556

print(10/3)
3.3333333333333335
```

We can also convert a floating point number to an integer or an integer to floating point number. Notice that Python by default rounds down when it converts from floating point to integer.

```
# convert a to integer
a = 7.83
int(a)
7

# convert to float
b = 7
float(b)
7.0
```

Working With Our Survey Data

Getting back to our data, we can modify the format of values within our data, if we want. For instance, we could convert the `record_id` field to floating point values.

```
# convert the record_id field from an integer to a float
surveys_df['record_id'] = surveys_df['record_id'].astype('float64')
surveys_df['record_id'].dtype
```

OUTPUT: `dtype('float64')`

Challenge - Changing Types

Try converting the column `plot_id` to floats using

```
surveys_df.plot_id.astype("float")
```

Next try converting `weight` to an integer. What goes wrong here? What is Pandas telling you? We will talk about some solutions to this later.

Missing Data Values - NaN

What happened in the last challenge activity? Notice that this throws a value error: `ValueError: Cannot convert NA to integer`. If we look at the `weight` column in the surveys data we notice that there are NaN (**N**ot **a** **N**umber) values. *NaN* values are undefined values that cannot be represented mathematically. Pandas, for example, will read an empty cell in a CSV or Excel sheet as a NaN. NaNs have some desirable properties: if we were to average the `weight` column without replacing our NaNs, Python would know to skip over those cells.

```
surveys_df['weight'].mean()
42.672428212991356
```

Dealing with missing data values is always a challenge. It's sometimes hard to know why values are missing - was it because of a data entry error? Or data that someone was unable to collect? Should the value be 0? We need to know how missing values are represented in the dataset in order to make good decisions. If we're lucky, we have some metadata that will tell us more about how null values were handled.

For instance, in some disciplines, like Remote Sensing, missing data values are often defined as -9999. Having a bunch of -9999 values in your data could really alter numeric calculations. Often in spreadsheets, cells are left empty where no data are available. Pandas will, by default, replace those missing values with NaN. However it is good practice to get in the habit of intentionally marking cells that have no data, with a no data value! That way there are no questions in the future when you (or someone else) explores your data.

Where Are the NaN's?

Let's explore the NaN values in our data a bit further. Using the tools we learned in lesson 02, we can figure out how many rows contain NaN values for weight. We can also create a new subset from our data that only contains rows with weight values > 0 (ie select meaningful weight values):

```
len(surveys_df[pd.isnull(surveys_df.weight)])
# how many rows have weight values?
len(surveys_df[surveys_df.weight> 0])
```

We can replace all NaN values with zeroes using the `.fillna()` method (after making a copy of the data so we don't lose our work):

```
df1 = surveys_df.copy()
# fill all NaN values with 0
df1['weight'] = df1['weight'].fillna(0)
```

However NaN and 0 yield different analysis results. The mean value when NaN values are replaced with 0 is different from when NaN values are simply thrown out or ignored.

```
df1['weight'].mean()
38.751976145601844
```

We can fill NaN values with any value that we chose. The code below fills all NaN values with a mean for all weight values.

```
df1['weight'] = surveys_df['weight'].fillna(surveys_df['weight'].mean())
```

We could also chose to create a subset of our data, only keeping rows that do not contain NaN values.

The point is to make conscious decisions about how to manage missing data. This is where we think about how our data will be used and how these values will impact the scientific conclusions made from the data.

Python gives us all of the tools that we need to account for these issues. We just need to be cautious about how the decisions that we make impact scientific results.

Challenge - Counting

Count the number of missing values per column. Hint: The method .count() gives you the number of non-NA observations per column. Try looking to the .isnull() method.

Recap

What we've learned:

- How to explore the data types of columns within a DataFrame
- How to change the data type
- What NaN values are, how they might be represented, and what this means for your work
- How to replace NaN values, if desired

Key Points

(../02-index-slice-subset/)

(../04-merging-data/)