[DS-04] Dates and strings in Python

Miguel-Angel Canela
Associate Professor, IESE Business School

Date and datetime

In computer environments, there are typically two data types for time, called date and datetime. In **type date** we can store dates, that is, year, month and day. Most software applications for data management and analysis can deal with different date formats. The default format for dates in most languages, including Python, is `yyyy-mm-dd`. I advise you to use only this, even in Excel, which chooses the date format based on the standard of the actual countries. Under the hood, a variable of type `date` is just a number, the number of days since the origin of time, which is, typically, 1970-01-01.

In data of **type datetime**, we store the same as in type date, plus hour, minute and second. The preferred format is `yyyy-mm-dd hh:mm:ss`. Sometimes, an indication of the time zone is added at the end, as we will see below. Examples are CET (Central European Time), CEST (Central European Summer Time), GMT (Greenwich Mean Time) and UTC (Coordinated Universal Time). Datetime is also called **timestamp**.

Dates can be managed in many ways in Python. The package `datetime` is recommended if you want to the deal with dates one by one, not within a data frame. The functions `datetime.date` and `datetime.datetime` can be used to create dates and datetimes. My presentation here is very brief, and restricted to the management of datetimes in Pandas data structures. So, I start by creating a series which contains a date as a string:

```
In [1]: import pandas as pd
   ...: date1 = pd.Series('2019-01-01')
   ...: date1
Out[1]:
0    2019-01-01
dtype: object
```

The function `to_datetime` transforms this string series into a datetime series:

```
In [2]: date2 = pd.to_datetime(date1)
   ...: date2
Out[2]:
0    2019-01-01
dtype: datetime64[ns]
```

We can go back to the original string type, with the method `astype`, which is typically used for conversions in Pandas structures:

```
In [3]: date2.astype(str)
Out[3]:
0    2019-01-01
dtype:  object
```

We can also see the data as a number of seconds since the origin:

```
In [4]: date2.astype(int)
Out[4]:
0    1546300800000000000
```

```
dtype:  int64
```

The method `apply` is used to apply a function term by term to a column of a data frame. `apply` is typically used in combination with **lambda functions**, that is, functions which are defined on the fly, not named, and forgotten after execution. Using `apply` and an appropriate lambda function, we can extract from a datetime series any information that can be extracted from a datatime variable. Example:

```
In [5]: date2.apply(lambda x:  x.month)
Out[5]:
0    1
dtype: int64
```

String data

There is a collection of methods for manipulating string variables. This includes not only sequences of **alphanumeric characters**, but also **white space** and **punctuation**. Beware that the **empty string** (`''`) is not the same as `nan`. It is a string of length zero.

We also have in Pandas a collection of methods which can be directly applied to series of type string. I present next a brief review of some useful methods.

- With method `str.len` we **get the length of every element of a string series**. Note in the following example how the empty string and the missing value are dealt with. Also, note that the series containing the lengths has been coerced to type float to cope with the `nan` value (the length has int type when all the strings have length).

```
In [6]: import numpy as np
In [7]: presidents = pd.Series(['Donald Trump', 'Bill Clinton',
   ...: '', np.nan])
   ...: presidents
Out[7]:
0    Donald Trump
1    Bill Clinton
2
3    NaN dtype: object
In [8]: presidents.str.len()
Out[8]:
0    12.0
1    12.0
2     0.0
3     NaN
dtype: float64
```

- **Substrings** can be extracted from a string variable just as we extract elements from a list:

```
In [9]: 'Charlie'[0:3]
Out[9]: 'Cha'
```

  The same works for a string series. This can be useful to manage dates, as shown in the following example.

```
In [10]: dates = pd.Series(['2016-10-06', '2015-08-19', '2016-01-30'])
    ...: dates.str[0:4]
Out[10]:
0    2016
1    2015
```

```
2    2016
dtype: object
```

- **Strings are joined** just as list, with the plus (+) sign:

```
In [11]: 'ab' + 'cd'
Out[11]: 'abcd'
```

The same works for a string series:

```
In [12]: firstnames = pd.Series(['Marvin', 'Leonard'])
    ...: secondnames = pd.Series(['Gaye', 'Cohen'])
    ...: firstnames + ' ' + secondnames
Out[12]:
0       Marvin Gaye
1     Leonard Cohen
dtype: object
```

- Many methods of string data analysis are based on counting the occurrences of selected terms. Counting is preceded by **conversion to lowercase**, performed with the method `str.lower`.

```
In [13]: students = pd.Series(['Pablo', 'Liudmila', 'Nana Yaa'])
    ...: students.str.lower()
Out[13]:
0       pablo
1    liudmila
2    nana yaa
dtype: object
```

- Method `str.contains` is used to **detect the presence or absence of a pattern in a string**. It returns a Boolean series indicating, term by term, whether the pattern occurs.

```
In [14]: students.str.contains('an')
Out[14]:
0    False
1    False
2     True
dtype: bool
```

- Method `str.findall` is used to **extract matching patterns from a string**. It produces, for each term of the series, a list containing all the occurrences of the pattern.

```
In [15]: students.str.findall('a')
Out[15]:
0             [a]
1             [a]
2    [a, a, a, a]
dtype: object
```

- With method `str.replace`, we can **replace matched patterns in a string**:

```
In [16]: students.str.replace(' ', '-')
Out[16]:
0       Pablo
1    Liudmila
2    Nana-Yaa
dtype: object
```

While the third argument of `str.replace` (the replacement) has to be a single string, the second argument (the pattern) can be multiple. In the preceding example, we replaced a single white space by a dash. Now, to replace either white space or the letter 'o', we set as the pattern to replace the regular expression 'o| '. Note that, in Python (as in many languages), the vertical bar means 'OR'.

```
In [17]: students.str.replace('o| ', '-')
Out[16]:
0       Pabl-
1    Liudmila
2    Nana-Yaa
dtype: object
```

- Method `str.split` **splits up a string into pieces**. This is one way to transform a string into a **bag of words**, that is, a list whose terms are the words contained in the string. For every term of a string series, the method returns the corresponding bag of words.

```
In [18]: sayings = pd.Series(['Correlation is not causation',
    ...: 'Flattery is the food of fools'])
    ...: sayings.str.split(' ')
Out[18]:
0        [Correlation, is, not, causation]
1    [Flattery, is, the, food, of, fools]
dtype: object
```

Regular expressions

Some of the transformations performed by the methods described in the preceding section are dramatically simplified by using **regular expressions**. A regular expression is a pattern which describes a set of strings.

Among the regular expressions, **character classes** are the simplest case. They are built by enclosing a collection of characters within square brackets. The square brackets indicate *any* of the characters enclosed. For instance, `[0-9]` stands for any digit, and `[A-Z]` for any capital letter.

I show how this works with some simple examples.

```
In [19]: bio = pd.Series(['I was born in 1954',
    ...: 'My phone is +34 932 534 200'])
    ...: bio.str.replace('[a-z]', 'x')
Out[19]:
0             I xxx xxxx xx 1954
1    Mx xxxxx xx +34 932 534 200
dtype: object
In [20]: bio.str.replace('[0-9]', 'x')
Out[20]:
0             I was born in xxxx
1    My phone is +xx xxx xxx xxx
dtype: object
```

Character classes get more powerful when complemented with **quantifiers**. For instance, followed by a plus sign (+), a character class indicates a sequence of any length. So, `[0-9]+` indicates any sequence of digits, therefore any number. We can also specify the minimum and maximum length of the sequence, as in the second example below.

```
In [21]: bio.str.replace('[a-zA-Z]+', 'x')
Out[21]:
```

```
0             x x x x 1954
1    x x x +34 932 534 200
dtype: object
In [22]: bio.str.replace('[0-9]{1,3}', 'x')
Out[22]:
0        I was born in xx
1    My phone is +x x x x
dtype: object
```

A simple clean way of getting a bag of words is as follows.

```
In [23]: bio.str.findall('[a-zA-Z0-9]+')
Out[23]:
0              [I, was, born, in, 1954]
1    [My, phone, is, 34, 932, 534, 200]
dtype: object
```

Regular expressions are a whole chapter of programming, with entire books, such as Friedl (2007), devoted to them. If you are interested, you may also try the `regexr` website at `www.regexr.com`, which makes fun of learning regular expressions.

Special characters

Text imported from PDF or HTML documents, or from devices like mobile phones, may contain **special characters** like the n-dash (–), the left/right quotation marks (“, ‘, etc), or the three-dot character (…), which is better to control, to avoid confusion. To keep this note short, I do not develop this point here, but mind that, if you capture text data on your own, you will probably find some of that in your data. Even if the documents are expected to be in English, they can be contaminated by other languages: Han characters, German umlaut, Spanish eñe, etc.

Another source of trouble is that these special symbols can be encoded by different computers or different text editors in different ways. The typical **encodings** in the Western world are UTF-8 and Latin-1. Python can deal with these two, but may have trouble with other encodings. I cannot say more, because this topic goes beyond my expertise, so I do not discuss encodings in this course. If you are interested, you may take a look at Korpela (2006).

References

1. JEF Friedl (2007), *Mastering Regular Expressions*, O’Reilly.
2. JK Korpela (2006), *Unicode Explained*, O’Reilly.
3. W McKinney (2017), *Python for Data Analysis — Data Wrangling with Pandas, NumPy, and IPython*, O’Reilly.