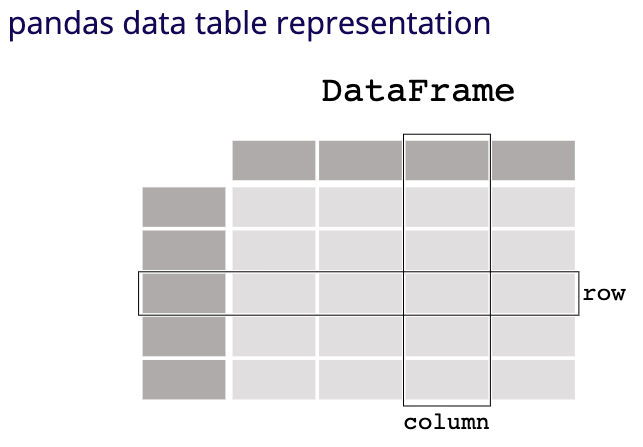
**Week 7 Pandas & regular expressions**

**Pandas**

Pandas is used in the data science process to get data and explore that data. Pandas provides data in a 2D table object called a DataFrame (similar to a spreadsheet).

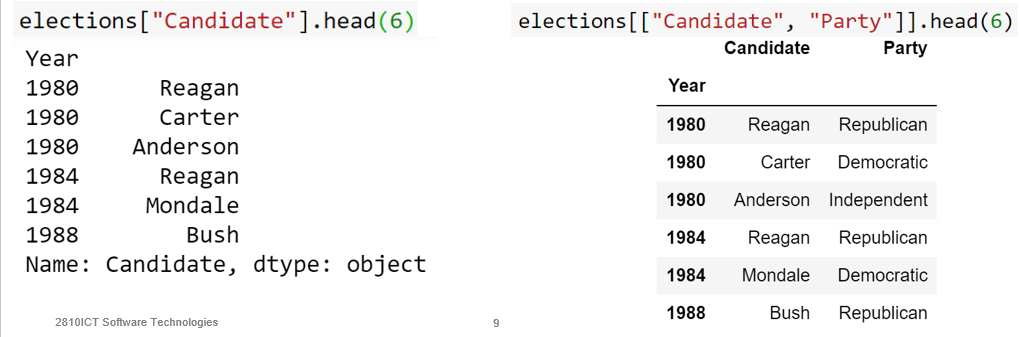
There are 3 fundamental data structures in Pandas

* DataFrame: 2D data, tabular data
* Series: 1D data, Column data
* Index: A sequence of row labels
* A DataFrame is a collection of series that all share the same index.
* Column names in Pandas are almost always unique (state, language, motto, etc)

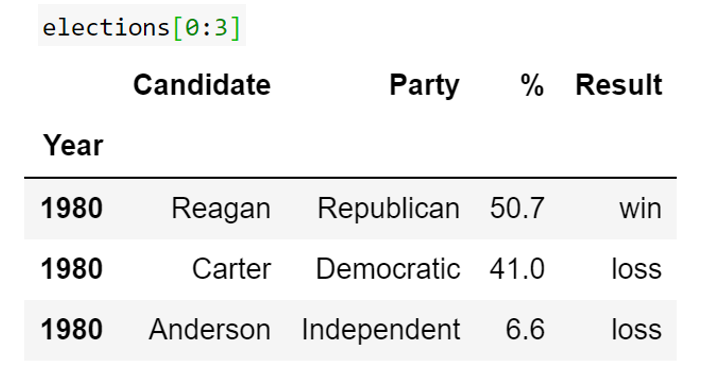
**Indexing**

**Indexing by column**

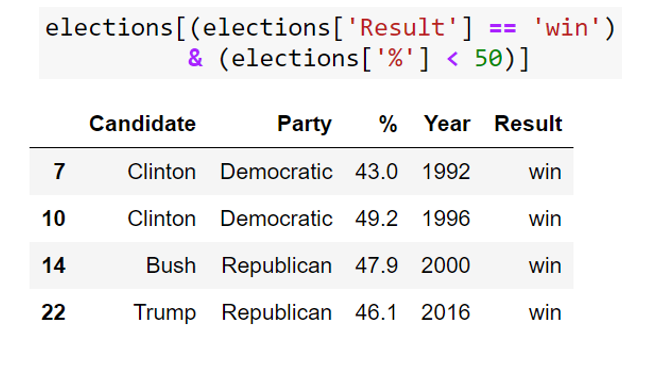
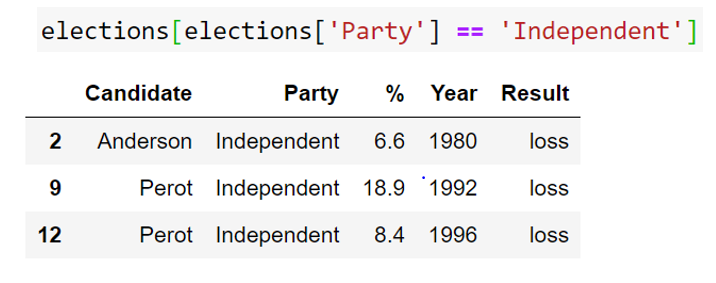
Indexing by column is the extraction of a series or collection of series.

* Column name argument yields series
* List argument yields a DataFrame (subset)
* 

**Indexing by row**

Indexing by row number slices yields a frame of those rows. Row indexing must use slices [n: m], indexing by a singular number will not yield the single row.

**Boolean indexing**

A Boolean array can be given for indexing also. This can be generated using logical operators on series. Boolean series can be combines using & allowi ng filtering of results by multiple critera.

**commands and functions**

**isin**

The ‘isin’ function allows for easy matching of possible row values.

* dataframe[(dataframe[“column”].isin([“value1”, “value2”)]

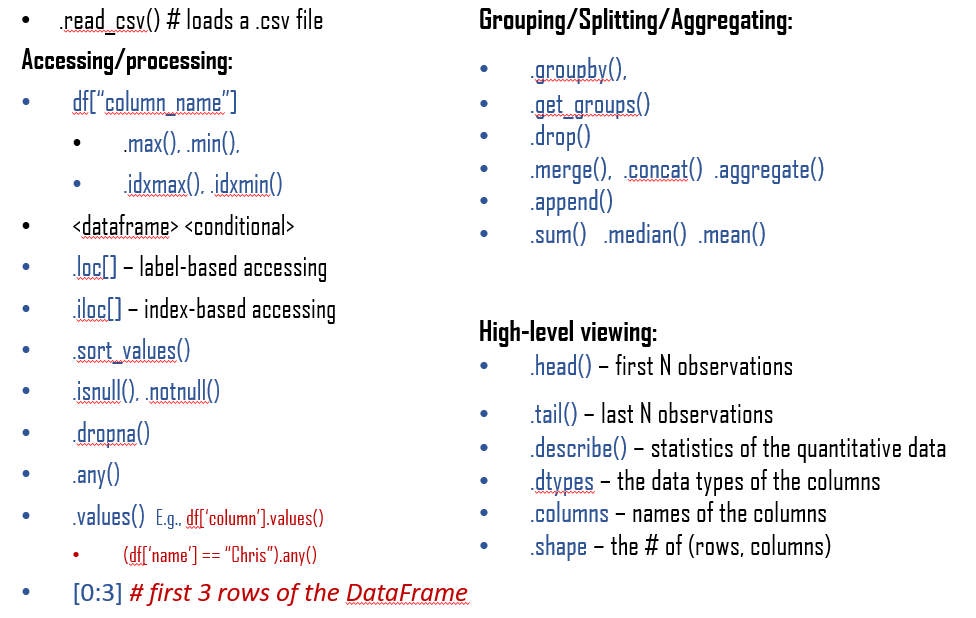
**Query**  
the query command privides an alternate way to combine multiple conditions.

* Tablename.query(“column == ‘value’ and column2 < value2”)

**Row selection**

In order to yield singular data rows, we can use the loc and iloc functions.

* Row label: Dataframe.loc[2]
* Row index: Dataframe.iloc[2]

**Other pandas functions**

**EDA – Exploratory Data Analysis**

1. Store data in data structure(s) that will be convenient for exploring/processing (Memory is fast. Storage is slow)
2. Clean/format the data so that:
   * Each row represents a single object/observation/entry
   * Each column represents an attribute/property/feature of that entry
   * Values are numeric whenever possible
   * Columns contain atomic properties that cannot be further decomposed
3. Explore global properties: use histograms, scatter plots, and aggregation functions to summarize the data
4. Explore group properties: group like-items together to compare subsets of the data (are the comparison results reasonable/expected?)

**Regular expressions**

Regular Expressions are used in programming languages to filter texts or text strings. It's possible to check, if a text or a string matches a regular expression. The syntax of regular expressions is the same for all programming and script languages, e.g. Python, Perl, Java, SED, AWK and even X#.

A basic example of using REs within Python is:

* >>>import re

>>>x = re.search("cat","A cat and a rat can't be friends.")

>>>print(x)

<\_sre.SRE\_Match object at 0x7fd4bf238238>

>>>x = re.search("cow","A cat and a rat can't be friends.")

>>>print(x)

None

**Syntax**

Special characters within square brackets:

* Square brackets, "[" and "]", are used to include a character class.
* For a choice between larger character classes the syntax of regular expressions supplies a meta-character "-". [a-e] a simplified writing for [abcde] or [0-5] denotes [012345].
* “^” used at the start of square brackets denotes ‘anything but’ what is contained in the bracket.
* [^0-9] = anything but a digit, [‘abc] = anything but a, b, or c

Special sequences

* \d = any decimal 0-9
* \D = any non-digit character ([^0-9])
* \s any white space character
* \w any alphanumeric character ([a-zA-Z0-9\_])
* \W anything but an alphanumeric character
* \b empty string at the start or end of a worf
* \B empty string that isn’t at the start or end of a word
* \\ matches a backslash

Functions

* Search(): checks for any string containing what is entered in the brackets.
  + Search(r”^expr”, str): checks for the expr at the start of the str
  + Can use multiline mode (re.MULTILINE)
* Match(): checks for a match of what is entered in the brackets.
  + Only for start of expression

Subexpressions

A subexpression is grouped by round brackets and a question mark following such a group means that this group may or may not exist. With the following expression we can match dates like "Feb 2011" or February 2011”: r"Feb(ruary)? 2011"

A star following a character or a subexpression group means that this expression or character may be repeated arbitrarily, even zero times. For example r"[0-9]\*" matches any sequence of digits, even the empty string while r".\*" matches any sequence of characters and the empty string.

Quantifier

A quantifier after a token, which can be a single character or group in brackets, specifies how often that preceding element is allowed to occur. The most common quantifiers are

the question mark ‘?’

the asterisk or star character \*, which is derived from the Kleene star

and the plus sign +, derived from the Kleene cross

Back references

The part of the string matched by the grouped part of the regular expression is stored in the back reference (i.e. the subexpression in parenthesis, is stored in a back reference).

Back references allow us to reuse parts of regular expressions. These stored vales can be reused inside the expression itself and afterwards.

Expressions

* span() returns a tuple with the start and end position
* The methods start() and end() are in a way superfluous as the information is contained in span()
* group(), if called without argument, returns the substring, which had been matched by the complete regular expression. With the help of group() we are also capable of accessing the matched substring by grouping parentheses, to get the matched substring of the n-th group, we call group() with the argument n: group(n).