Data 6 Python Cheat Sheet

This cheat sheet has been modified from the Data 6 Python Reference and includes all of the functions and table methods that you will need for the exams.

Built-In Python Functions

Function	Input	Description/Output
`str(val)`	A value of any type (`int`, `float`, `NoneType`, etc.)	Converts `val` to a string
`int(num)`, `float(num)`	A numerical value (represented as a `string` or `float`)	Converts `num` to the respective numeric type
`len(seq)`	`array` or `list` or `string`	Returns the length of `seq`
`max(arr)`, `min(arr)`, `sum(arr)`	`array` or `list`	Returns a value in `arr`. Respectively: maximum, minimum, sum.
`abs(val)`	`int` or `float` or `array`	Returns the absolute value of `val`. If `val` is an array, return an array of absolute values.
`round(num)`, `round(num, ndigits)`	`int` or `float`	Returns `num` rounded to the nearest integer. If optional `ndigits` is provided, round to `ndigits` precision after the decimal point.
`print(input, ...)`	`input`: any inputs to print	Prints the input values, separated by spaces
`type(object)`	`object`: the object whose type is to be determined	Returns the type of `object`

NumPy Array Functions

Function	Description
`make_array(val1, val2, ...)`	Makes a NumPy array with the inputted values
`np.mean(arr)` or `np.average(arr)`	Calculates the average value of `arr`
`np.sum(arr)`	Returns the sum of the values in `arr`
`np.prod(arr)`	Returns the product of the values in `arr`
`np.sqrt(num)`	Calculates the square root of num
`np.arange(stop)`, `np.arange(start, stop)`, or `np.arange(start, stop, step)`	Creates an array of sequential numbers starting at `start`, going up in increments of `step`, and going up to but excluding `stop`. When `start` and/or `step` are not specified, default values are used in their place. Default `start` is 0, default `step` is 1.
`np.count_nonzero(arr)`	Returns the number of non-zero (or True) elements in `arr`.
`np.append(arr, item)`	Appends item to the end of `arr`. Does not modify the original array.
`np.cumsum(arr)`	Returns the cumulative sum of the elements in `arr`, where each element is the sum of all preceding elements including itself
`np.diff(arr)`	Returns a new array of size `len(arr)-1` with elements equal to the difference between adjacent elements; `val_2 - val_1`, `val_3 - val_2`, etc.
`np.sort(arr)`	Sorts an array in ascending order. This function returns nothing; array `arr` is sorted in-place.
`arr.item(i)`	[method] Returns the `i`-th item in `arr` (remember Python indices start at 0!).
`arr.take(indices)`	[method] Creates an array with only the elements of `arr` at the given indices. indices is either an array of indices or an integer corresponding to one index.

`percentile(p, arr)`	Returns the corresponding percentile `p` of an array `arr`.

String Methods and Operations

Function	Description
`s.upper()`	Returns a copy of `s` where all letters are uppercase.
`s.lower()`	Returns a copy of `s` where all letters are lowercase.
`s.replace(old, new)`	Returns a copy of `s` with all occurrences of the substring `old` replaced by new.
`s.split()` or `s.split(separator)` or `s.split(separator, maxsplit)`	Splits `s` into a list of substrings using the specified `separator`; returns a list of strings. If `separator` is not provided, splits at any whitespace. You can also use the optional argument `maxsplit` to limit the number of splits.
`s.join(iterable)`	Returns a string that concatenates the elements in `iterable` (usually a list or array) into a single string, with each element separated by `s`.
`s.strip(chars)`	Returns a copy of `s` with the characters in `chars` trimmed off the left and right ends of the string. If `chars` is not provided, trims off whitespace.
`substr in s`	Returns `True` if the string `substr` is contained in `s` (i.e., `substr` is a substring of `s`) and `False` otherwise. A similar operation, `x in lst`, checks if an element x is in an iterable (list or array).
`substr not in s`	Returns `True` if `substr` is not a substring of `s` and `False` otherwise. A similar operation, `x not in lst`, checks if an element `x` is not in an iterable (list or array).

Tables and Table Methods

Function	Description	Input	Output
`Table()`	Creates an empty table, usually to extend with data	None	**Table**: An empty Table
`tbl.with_column(name, values)` or `tbl.with_columns(n1, v1, n2, v2, ...)`	Adds an extra column onto `tbl` with the label `name` and `values` as the column values	1. `string`: name of the new column 2. `array`: values in the column	**Table**: a copy of the original table with the new column(s)
`tbl.column(col)`	Returns the values in a column in `tbl`	`string` or `int`: the column name or index	**array**: the values in that column
`tbl.num_rows`	Compute the number of rows in `tbl`	None	**int**: the number of rows in the table
`tbl.num_columns`	Compute the number of columns in `tbl`	None	**int**: the number of columns in the table
`tbl.labels`	Returns the labels in `tbl`	None	**array**: the names of each column as strings
`tbl.select(col1, col2, ...)`	Creates a copy of `tbl` only with the selected columns	`string` or `int`: column name(s) or index(es)	**Table** with the selected columns

`tbl.drop(col1, col2, ...)`	Creates a copy of `tbl` without the selected columns	`string` or `int`: column name(s) or index(es)	**Table** without the selected columns
`tbl.relabeled(old_label, new_label)`	Creates a new table, changing the column name specified by `old_label` to `new_label`, and leaves the original table unchanged	1. `string`: old column name 2. `string`: new column name	**Table**: a copy of the original table with the changed column name
`tbl.show(n)`	Displays the first `n` rows of `tbl`. Defaults to showing the entire table	(Optional) `int`: number of rows to display	**None** (table is displayed)
`tbl.sort(column_name[, descending=True])`	Sorts rows by `column_name`. Ascending by default unless `descending=True` is included	1. `string` or `int`: name or index of the column to sort 2. (Optional) `descending=True`	**Table**: a copy of the original table with the column sorted
`tbl.where(column, predicate)`	Creates a copy of `tbl` containing only the rows where the value of `column` matches the `predicate`	1. `string` or `int`: column 2. `are.(...)` predicate	**Table**: a copy of the original table with only the rows that match the predicate
`tbl.take(row_indices)`	Creates a table with only the rows at the given indices	`int` or `array`: indices of rows to be included in the table	**Table**: a copy of the original table with only the rows at the given indices
`tbl.apply(function)` or `tbl.apply(function, col1, col2, ...)`	Returns an array of values resulting from applying a function to each item in a column.	1. Function: function to apply to column 2. (Optional) `string` or `i`: the column name(s) or index(es) to apply the function to	**array** containing an element for each value in the original column after applying the function to it
`tbl.group( column_or_columns, function)`	Groups rows in `tbl` by unique values or combinations of values in a column(s). Multiple columns must be entered as an array of strings. Values in the other columns are aggregated by count (by default) or the optional argument `function`.	1. `string` or array of `string`s: column(s) on which to group 2. (Optional) Function: function to aggregate values in cells (defaults to counting rows)	**Table** a new groupped table

`tbl.pivot(col1, col2)` or `tbl.pivot(col1, col2, values, collect)`	Creates a pivot table where each unique value in `col1` has its own column and each unique value in `col2` has its own row. Counts or aggregates values from a third column, collected with some function. If the values and collect arguments are not included, pivot defaults to returning counts in the cells.	1. `string`: name of the column in `tbl` whose unique values will make up the columns of the pivot table 2. `string`: name of column in `tbl` whose unique values will make up the rows of the pivot table 3. (Optional) `string`: name of the column in tbl that describes the values of cells in the pivot table 4. (Optional) Function: how the values are collected (e.g. `sum` or `np.mean`)	**Table**: a new pivot table
`tblA.join(colA, tblB)` or `tblA.join(colA, tblB, colB)`	Generate a table with the columns of `tblA` and `tblB`, containing rows for all values in `colA` and `colB` that appear in `tblA` and `tblB`, respectively. By default, `colB` is the same value as `colA`. `colA` and `colB` must be strings specifying column names.	1. `string`: name of column in `tblA` with values to join on 2. **Table**: the other table 3. (Optional) `string`: the name of the shared column in `tblB`, if column names are different between the tables	**Table**: a new combined table
`tbl.with_row(values)`	Adds a new row with the specified `values` to `tbl`	**list or array**: values to add as a new row	**Table**: a copy of the original table with the new row
`tbl.with_rows( list_of_rows)`	Adds multiple rows to `tbl` using a list of rows	**list of lists or arrays**: each list/array represents a new row	**Table**: a copy of the original table with the new rows

Visualization Functions

Function	Description	Input	Output
`tbl.barh(categories)` or `tbl.barh(categories, values)`	Displays a horizontal bar chart with bars for each category in the column `categories` `values` specifies the column corresponding to the size of each bar, but is unnecessary if the table only has two columns. Optional argument `overlay` (default is `True`) specifies whether grouped bar charts should be overlaid or on separate plots.	1. `string`: name of the column with categories 2. (Optional) `string`: name of the column with values corresponding to the categories	**None**: draws a bar chart
`tbl.hist(column)`	Generates a histogram of the numerical values in `column`. Optional arguments `group` (to specify categorical column to group on), `bins` (to specify custom bins), and `overlay` to specify overlaid or separate histograms.	`string`: name of the column	**None**: draws a histogram

`tbl.plot(x_column, y_column)` or `tbl.plot(x_column)`	Draws a line plot consisting of one point for each row in `tbl`. If only `x_column` is specified, `plot` will plot the rest of the columns on the y-axis with different colored lines. Optional argument `overlay` (default is True) specifies whether multiple lines should be overlaid or on separate plots	1. `string`: name of the column for x-axis 2. `string`: name of the column for y-axis	**None**: draws a line plot
`tbl.scatter(x_column, y_column)`	Draws a scatter plot consisting of one point for each row in `tbl`. The optional argument `fit_line=True` can be included to draw a line of best fit through the scatter plot. The optional arguments `group` (to specify categorical column to group on) and `sizes` (to specify a numerical column for bubble sizes) can also be used to encode additional variables.	1. `string`: name of the column on the x-axis 2. `string`: name of the column on the y-axis 3. (Optional) `fit_line=True`	**None**: draws a scatter plot

Table.where (Boolean) Predicates

These functions can be passed in as the second argument to `tbl.where(..)` and act as a condition by which to select rows from `tbl`.

Predicate	**Description**
`are.equal_to(Z)`	Equal to `Z` (`Z` can be an `int`, `float`, or `string`).
`are.not_equal_to(Z)`	Not equal to `Z` (`Z` can be an `int`, `float`, or `string`).
`are.above(x)`	Greater than `x`.
`are.above_or_equal_to(x)`	Greater than or equal to `x`.
`are.below(x)`	Less than `x`.
`are.below_or_equal_to(x)`	Less than or equal to `x`.
`are.between(x, y)`	Greater than or equal to `x` and less than `y`.
`are.between_or_equal_to(x, y)`	Greater than or equal to `x` and less than or equal to `y`.
`are.strictly_between(x, y)`	Greater than `x` and less than `y`.
`are.contained_in(A)`	True if it is a substring of `A` (if `A` is a **string**) or an element of `A` (if `A` is an array)
`are.containing(S)`	Contains the string `S`.

Iterating over dictionaries with a `for` loop

Lecture Notes printout. Five code cells with corresponding output.

```
slang = {
    'smh': 'shake my head',
    'lol': 'laugh out loud',
    'GOAT': 'greatest of all time'
}
slang
```

```
{'smh': 'shake my head',
 'lol': 'laugh out loud',
 'GOAT': 'greatest of all time'}
```

```
for abb in slang:
    print(abb, ":", slang[abb])
```

```
smh : shake my head
lol : laugh out loud
GOAT : greatest of all time
```

```
for abb in slang.keys():
    print(abb, ":", slang[abb])
```

```
smh : shake my head
lol : laugh out loud
GOAT : greatest of all time
```

```
for slang_phrase in slang.values():
    print(slang_phrase)
```

```
shake my head
laugh out loud
greatest of all time
```

```
for k, v in slang.items():
    print(k, ":", v)
```

```
smh : shake my head
lol : laugh out loud
GOAT : greatest of all time
```