**Python for Data Science For Dummies Cheat Sheet**

From [Python for Data Science For Dummies](http://www.dummies.com/store/product/Python-for-Data-Science-For-Dummies.productCd-1118843983.html)

By [John Paul Mueller,](http://www.dummies.com/?s=&a=john-paul-mueller) [Luca Massaron](http://www.dummies.com/?s=&a=luca-massaron)

Python is an incredible programming language that you can use to perform data science tasks with minimal effort. The huge number of available libraries means that the low-level code you normally need to write is likely already available from some other source. All you need to focus on is getting the job done. With that in mind, this cheat sheet helps you access the most commonly needed reminders for making your programming experience fast and easy.

**The 8 Most Common Python Programming Errors**

Developers everywhere make errors at times. However, you might be able to save some time and work if you know about the most frequent types of programming errors that people make with Python. The following list tells you about these common mistakes:

* **Having the incorrect indentation:** Many Python features rely on indentation. For example, when you create a new class, everything in that class is indented under the class declaration. The same is true for decision, loop, and other structural statements. If you find that your code is executing a task when it really shouldn’t, start reviewing the indentation you’re using.
* **Using the assignment operator instead of the equality operator:** When performing a comparison between two objects or value, you just use the equality operator (==), not the assignment operator (=). The assignment operator places an object or value within a variable and doesn’t compare anything.
* **Putting function calls in the wrong order when creating complex statements:** Python always executes functions from left to right. So the statement MyString.strip().center(21, “\*”) produces a different result than MyString.center(21, “\*”).strip(). When you encounter a situation in which the output of a series of concatenated functions is different from what you expected, you need to check function order to ensure that each function is in the correct place.
* **Misplacing punctuation:** It’s possible to put punctuation in the wrong place and create an entirely different result. Remember that you must include a colon at the end of each structural statement. In addition, parentheses placement is critical. For example, (1 + 2) \* (3 + 4), 1 + ((2 \* 3) + 4), and 1 + (2 \* (3 + 4)) all produce different results.
* **Using the incorrect logical operator:** Most of the operators don’t present developers with problems, but the logical operators do. Remember to use and to determine when both operands must be True and or when either of the operands can be True.
* **Creating count-by-one errors on loops:** Remember that a loop doesn’t count the last number you specify in a range. So if you specify the range [1:11], you actually get output for values between 1 and 10.
* **Having the wrong capitalization:** Python is case sensitive, so MyVar is different from myvar and MYVAR. Always check capitalization when you find that you can’t access a value you expected to access.
* **Spelling something wrong:** Even seasoned developers suffer from spelling errors at times. Ensuring that you use a common approach to naming variables, classes, and functions does help. However, even a consistent naming scheme won’t always prevent you from typing MyVer when you meant to type MyVar.

**Line Plot Styles**

Whenever you create a plot in Python, you need to identify the sources of information using more than just the lines. Creating a plot that uses differing line types and data point symbols makes the plot much easier for other people to use. The following table lists the line plot styles.

|  |  |  |
| --- | --- | --- |
| **Color** | **Marker** | **Style** |
| **Code** | **Line Color** | **Code** | **Marker Style** | **Code** | **Line Style** |
| b | blue | . | point | – | Solid |
| g | green | o | circle | : | Dotted |
| r | red | x | x-mark | -. | dash dot |
| c | cyan | + | plus | — | Dashed |
| m | magenta | \* | star | (none) | no line |
| y | yellow | s | square |  |  |
| k | black | d | diamond |  |  |
| w | white | v | down triangle |  |  |
|  |  | ^ | up triangle |  |  |
|  |  | < | left triangle |  |  |
|  |  | > | right triangle |  |  |
|  |  | p | 5-point star |  |  |
|  |  | h | 6-point star |  |  |

Remember that you can also use these styles with other kinds of plots. For example, a scatter plot can use these styles to define each of the data points. When in doubt, try the styles out to see whether they’ll work with your particular plot.

**Common IPython Magic Functions**

It’s kind of amazing to think that IPython provides you with magic, but that’s precisely what you get with the magic functions. A magic function begins with either a % or %% sign. Those with a % sign work within the environment, and those with a %% sign work at the cell level.

The following list provides you with a few of the most common magic functions and their purpose. To obtain a full listing, type **%quickref** and press Enter in the IPython console or check out the [full listing](https://damontallen.github.io/IPython-quick-ref-sheets/).

|  |  |  |
| --- | --- | --- |
| **Magic Function** | **Type Alone Provides Status?** | **Description** |
| %%timeit | No | Calculates the best time performance for all the instructions in a cell, apart from the one placed on the same cell line as the cell magic (which could therefore be an initialization instruction). |
| %%writefile | No | Writes the contents of a cell to the specified file. |
| %alias | Yes | Assigns or displays an alias for a system command. |
| %autocall | Yes | Makes it possible to call functions without including the parentheses. The settings are Off, Smart (default), and Full. The Smart setting applies the parentheses only if you include an argument with the call. |
| %automagic | Yes | Makes it possible to call the line magic functions without including the % sign. The settings are False (default) and True. |
| %cd | Yes | Changes directory to a new storage location. You can also use this command to move through the directory history or to change directories to a bookmark. |
| %cls | No | Clears the screen. |
| %colors | No | Specifies the colors used to display text associated with prompts, the information system, and exception handlers. You can choose between NoColor (black and white), Linux (default), and LightBG. |
| %config | Yes | Makes it possible to configure IPython. |
| %dhist | Yes | Displays a list of directories visited during the current session. |
| %file | No | Outputs the name of the file that contains the source code for the object. |
| %hist | Yes | Displays a list of magic function commands issued during the current session. |
| %install\_ext | No | Installs the specified extension. |
| %load | No | Loads application code from another source, such as an online example. |
| %load\_ext | No | Loads a Python extension using its module name. |
| %lsmagic | Yes | Displays a list of the currently available magic functions. |
| %matplotlib | Yes | Sets the backend processor used for plots. Using the inline value displays the plot within the cell for an IPython Notebook file. The possible values are: gtk’, ‘gtk3’, ‘inline’, ‘nbagg’, ‘osx’, ‘qt’, ‘qt4’, ‘qt5’, ‘tk’, and ‘wx’. |
| %paste | No | Pastes the content of the clipboard into the IPython environment. |
| %pdef | No | Shows how to call the object (assuming that the object is callable). |
| %pdoc | No | Displays the docstring for an object. |
| %pinfo | No | Displays detailed information about the object (often more than provided by help alone). |
| %pinfo2 | No | Displays extra detailed information about the object (when available). |
| %reload\_ext | No | Reloads a previously installed extension. |
| %source | No | Displays the source code for the object (assuming that the source is available). |
| %timeit | No | Calculates the best performance time for an instruction. |
| %unalias | No | Removes a previously created alias from the list. |
| %unload\_ext | No | Unloads the specified extension. |

**Scikit-Learn Method Summary**

Scikit-learn is a focal point for data science work with Python, so it pays to know which methods you need most. The following list gives you a brief overview of the most important methods used for data analysis.

* feature\_extraction.FeatureHasher

**Usage:** Preparing your data

**Description:** The hashing trick, allowing you to accommodate a large number of features in your dataset

* preprocessing.Binarizer

**Usage:** Preparing your data

**Description:** Create binary variables (feature values to 0 or 1)

* preprocessing.Imputer

**Usage:** Preparing your data

**Description:** Missing values imputation

* preprocessing.MinMaxScaler

**Usage:** Preparing your data

**Description:** Create variables bound by a minimum and maximum value

* preprocessing.OneHotEncoder

**Usage:** Preparing your data

**Description:** Transform categorical integer features into binary ones

* preprocessing.StandardScaler

**Usage:** Preparing your data

**Description:** Variable standardization by removing the mean and scaling to unit variance

* feature\_extraction.text.CountVectorizer

**Usage:** Preparing your data

**Description:** Convert text documents into a matrix of count data

* feature\_extraction.text.HashingVectorizer

**Usage:** Preparing your data

**Description:** Directly convert your text using the hashing trick

* feature\_extraction.text.TfidfVectorizer

**Usage:** Preparing your data

**Description:** Creates a dataset of TF-IDF features.

* feature\_selection.RFECV

**Usage:** Feature selection

**Description:** Automatic feature selection

* decomposition.PCA

**Usage:** Dimensionality reduction

**Description:** Principal component analysis (PCA)

* decomposition.RandomizedPCA

**Usage:** Dimensionality reduction

**Description:** Principal component analysis (PCA) using randomized SVD

* cross\_validation.cross\_val\_score

**Usage:** Cross-validation phase

**Description:** Estimate the cross-validation score

* cross\_validation.KFold

**Usage:** Cross-validation phase

**Description:** Divide the dataset into k folds for cross validation

* cross\_validation.StratifiedKFold

**Usage:** Cross-validation phase

**Description:** Stratified validation that takes into account the distribution of the classes you predict

* cross\_validation.train\_test\_split

**Usage:** Cross-validation phase

**Description:** Split your data into training and test sets

* grid\_search.GridSearchCV

**Usage:** Optimization

**Description:** Exhaustive search in order to maximize a machine learning algorithm

* linear\_model.LinearRegression

**Usage:** Prediction

**Description:** Linear Regression

* linear\_model.LogisticRegression

**Usage:** Prediction

**Description:** Linear Logistic Regression

* neighbors.KNeighborsClassifier

**Usage:** Prediction

**Description:** K-Neighbors classification

* naive\_bayes.MultinomialNB

**Usage:** Prediction

**Description:** Multinomial Naïve Bayes

* metrics.accuracy\_score

**Usage:** Solution evaluation

**Description:** Accuracy classification score.

* metrics.f1\_score

**Usage:** Solution evaluation

**Description:** Compute the F1 score, balancing accuracy and recall

* metrics.mean\_absolute\_error

**Usage:** Solution evaluation

**Description:** Mean absolute error regression error

* metrics.mean\_squared\_error

**Usage:** Solution evaluation

**Description:** Mean squared error regression error

* metrics.roc\_auc\_score

**Usage:** Solution evaluation

**Description:** Compute Area Under the Curve (AUC) from prediction scores