

# Python For Data Science Cheat Sheet

## Python Basics

Learn More Python for Data Science Interactively at [www.datacamp.com](https://www.datacamp.com)



### Variables and Data Types

#### Variable Assignment

```
>>> x=5
>>> x
5
```

#### Calculations With Variables

>>> x+2 7	Sum of two variables
>>> x-2 3	Subtraction of two variables
>>> x*2 10	Multiplication of two variables
>>> x**2 25	Exponentiation of a variable
>>> x%2 1	Remainder of a variable
>>> x/float(2) 2.5	Division of a variable

#### Types and Type Conversion

str()	'5', '3.45', 'True'	Variables to strings
int()	5, 3, 1	Variables to integers
float()	5.0, 1.0	Variables to floats
bool()	True, True, True	Variables to booleans

### Asking For Help

```
>>> help(str)
```

### Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

#### String Operations

```
>>> my_string * 2
'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
'thisStringIsAwesomeInnit'
>>> 'm' in my_string
True
```

### Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

#### Selecting List Elements

Index starts at 0

##### Subset

```
>>> my_list[1]
>>> my_list[-3]
```

Select item at index 1  
Select 3rd last item

##### Slice

```
>>> my_list[1:3]
>>> my_list[1:]
>>> my_list[:3]
>>> my_list[:]
```

Select items at index 1 and 2  
Select items after index 0  
Select items before index 3  
Copy my\_list

##### Subset Lists of Lists

```
>>> my_list2[1][0]
>>> my_list2[1][:2]
```

my\_list[list][itemOfList]

#### List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

#### List Methods

>>> my_list.index(a)	Get the index of an item
>>> my_list.count(a)	Count an item
>>> my_list.append('!')	Append an item at a time
>>> my_list.remove('!')	Remove an item
>>> del(my_list[0:1])	Remove an item
>>> my_list.reverse()	Reverse the list
>>> my_list.extend('!')	Append an item
>>> my_list.pop(-1)	Remove an item
>>> my_list.insert(0, '!')	Insert an item
>>> my_list.sort()	Sort the list

#### String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

#### String Methods

>>> my_string.upper()	String to uppercase
>>> my_string.lower()	String to lowercase
>>> my_string.count('w')	Count String elements
>>> my_string.replace('e', 'i')	Replace String elements
>>> my_string.strip()	Strip whitespaces

### Libraries

#### Import libraries

```
>>> import numpy
>>> import numpy as np
Selective import
>>> from math import pi
```

pandas Data analysis	Machine learning
NumPy Scientific computing	matplotlib 2D plotting

### Install Python

ANACONDA Leading open data science platform powered by Python	spyder Free IDE that is included with Anaconda	jupyter Create and share documents with live code, visualizations, text, ...
---	--	---

### Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3], [4,5,6]])
```

#### Selecting Numpy Array Elements

Index starts at 0

##### Subset

```
>>> my_array[1]
2
```

Select item at index 1

##### Slice

```
>>> my_array[0:2]
array([1, 2])
```

Select items at index 0 and 1

##### Subset 2D Numpy arrays

```
>>> my_2darray[:,0]
array([1, 4])
my_2darray[rows, columns]
```

#### Numpy Array Operations

```
>>> my_array > 3
array([False, False, False,  True], dtype=bool)
>>> my_array * 2
array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
array([6, 8, 10, 12])
```

#### Numpy Array Functions

>>> my_array.shape	Get the dimensions of the array
>>> np.append(other_array)	Append items to an array
>>> np.insert(my_array, 1, 5)	Insert items in an array
>>> np.delete(my_array, [1])	Delete items in an array
>>> np.mean(my_array)	Mean of the array
>>> np.median(my_array)	Median of the array
>>> my_array.corrcoef()	Correlation coefficient
>>> np.std(my_array)	Standard deviation



# Python For Data Science Cheat Sheet

## Importing Data

Learn Python for data science [Interactively](#) at [www.DataCamp.com](#)



### Importing Data in Python

Most of the time, you'll use either NumPy or pandas to import your data:

```
>>> import numpy as np
>>> import pandas as pd
```

### Help

```
>>> np.info(np.ndarray.dtype)
>>> help(pd.read_csv)
```

### Text Files

#### Plain Text Files

```
>>> filename = 'huck_finn.txt'
>>> file = open(filename, mode='r')
>>> text = file.read()
>>> print(file.closed)
>>> file.close()
>>> print(text)
```

Open the file for reading  
Read a file's contents  
Check whether file is closed  
Close file

Using the context manager with

```
>>> with open('huck_finn.txt', 'r') as file:
    print(file.readline())
    print(file.readline())
    print(file.readline())
```

Read a single line

#### Table Data: Flat Files

#### Importing Flat Files with numpy

##### Files with one data type

```
>>> filename = 'mnist.txt'
>>> data = np.loadtxt(filename,
    delimiter=',',
    skiprows=2,
    usecols=[0,2],
    dtype=str)
```

String used to separate values  
Skip the first 2 lines  
Read the 1st and 3rd column  
The type of the resulting array

##### Files with mixed data types

```
>>> filename = 'titanic.csv'
>>> data = np.genfromtxt(filename,
    delimiter=',',
    names=True,
    dtype=None)
```

Look for column header

```
>>> data_array = np.recfromcsv(filename)

The default dtype of the np.recfromcsv() function is None.
```

#### Importing Flat Files with pandas

```
>>> filename = 'winequality-red.csv'
>>> data = pd.read_csv(filename,
    nrows=5,
    header=None,
    sep='\t',
    comment='#',
    na_values=[""])
```

Number of rows of file to read  
Row number to use as col names  
Delimiter to use  
Character to split comments  
String to recognize as NA/NaN

### Excel Spreadsheets

```
>>> file = 'urbanpop.xlsx'
>>> data = pd.ExcelFile(file)
>>> df_sheet2 = data.parse('1960-1966',
    skiprows=[0],
    names=['Country',
           'AAM: War(2002)'])

>>> df_sheet1 = data.parse(0,
    parse_cols=[0],
    skiprows=[0],
    names=['Country'])
```

To access the sheet names, use the sheet\_names attribute:

```
>>> data.sheet_names
```

### SAS Files

```
>>> from sas7bdat import SAS7BDAT
>>> with SAS7BDAT('urbanpop.sas7bdat') as file:
    df_sas = file.to_data_frame()
```

### Stata Files

```
>>> data = pd.read_stata('urbanpop.dta')
```

### Relational Databases

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://Northwind.sqlite')
```

Use the table\_names() method to fetch a list of table names:

```
>>> table_names = engine.table_names()
```

#### Querying Relational Databases

```
>>> con = engine.connect()
>>> rs = con.execute("SELECT * FROM Orders")
>>> df = pd.DataFrame(rs.fetchall())
>>> df.columns = rs.keys()
>>> con.close()
```

Using the context manager with

```
>>> with engine.connect() as con:
    rs = con.execute("SELECT OrderID FROM Orders")
    df = pd.DataFrame(rs.fetchmany(size=5))
    df.columns = rs.keys()
```

#### Querying relational databases with pandas

```
>>> df = pd.read_sql_query("SELECT * FROM Orders", engine)
```

### Exploring Your Data

#### NumPy Arrays

```
>>> data_array.dtype
>>> data_array.shape
>>> len(data_array)
```

Data type of array elements  
Array dimensions  
Length of array

#### pandas DataFrames

```
>>> df.head()
>>> df.tail()
>>> df.index
>>> df.columns
>>> df.info()
>>> data_array = data.values
```

Return first DataFrame rows  
Return last DataFrame rows  
Describe index  
Describe DataFrame columns  
Info on DataFrame  
Convert a DataFrame to an NumPy array

### Pickled Files

```
>>> import pickle
>>> with open('pickled_fruit.pkl', 'rb') as file:
    pickled_data = pickle.load(file)
```

### HDF5 Files

```
>>> import h5py
>>> filename = 'H-H1_LOSC_4_v1-815411200-4096.hdf5'
>>> data = h5py.File(filename, 'r')
```

### Matlab Files

```
>>> import scipy.io
>>> filename = 'workspace.mat'
>>> mat = scipy.io.loadmat(filename)
```

### Exploring Dictionaries

#### Accessing Elements with Functions

```
>>> print(mat.keys())
>>> for key in data.keys():
    print(key)
```

Print dictionary keys  
Print dictionary keys

```
meta
quality
strain
```

```
>>> pickled_data.values()
>>> print(mat.items())
```

Return dictionary values  
Returns items in list format of (key, value) tuple pairs

#### Accessing Data Items with Keys

```
>>> for key in data['meta'].keys():
    print(key)
```

Explore the HDF5 structure

```
Description
DescriptionURL
Detector
Duration
GPSstart
Observatory
Type
UTCstart
```

```
>>> print(data['meta']['Description'].value)
```

Retrieve the value for a key

### Navigating Your FileSystem

#### Magic Commands

```
!ls
%cd ..
%pwd
```

List directory contents of files and directories  
Change current working directory  
Return the current working directory path

#### os Library

```
>>> import os
>>> path = "/usr/tmp"
>>> wd = os.getcwd()
>>> os.listdir(wd)
>>> os.chdir(path)
>>> os.rename("test1.txt",
    "test2.txt")
>>> os.remove("test1.txt")
>>> os.mkdir("newdir")
```

Store the name of current directory in a string  
Output contents of the directory in a list  
Change current working directory  
Rename a file  
Delete an existing file  
Create a new directory



# Python For Data Science Cheat Sheet

## NumPy Basics

Learn Python for Data Science Interactively at [www.DataCamp.com](https://www.datacamp.com)



### NumPy

The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:



NumPy

```
>>> import numpy as np
```

### NumPy Arrays

#### 1D array

```
1 2 3
```

#### 2D array

axis 1  
axis 0

```
1.5 2 3  
4 5 6
```

#### 3D array

axis 2  
axis 1  
axis 0

### Creating Arrays

```
>>> a = np.array([1,2,3])  
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)  
>>> c = np.array([(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],  
                dtype = float)
```

### Initial Placeholders

```
>>> np.zeros((3,4))  
>>> np.ones((2,3,4),dtype=np.int16)  
>>> d = np.arange(10,25,5)  
  
>>> np.linspace(0,2,9)  
  
>>> e = np.full((2,2),7)  
>>> f = np.eye(2)  
>>> np.random.random((2,2))  
>>> np.empty((3,2))
```

Create an array of zeros  
Create an array of ones  
Create an array of evenly spaced values (step value)  
Create an array of evenly spaced values (number of samples)  
Create a constant array  
Create a 2X2 identity matrix  
Create an array with random values  
Create an empty array

### I/O

#### Saving & Loading On Disk

```
>>> np.save('my_array', a)  
>>> np.savez('array.npz', a, b)  
>>> np.load('my_array.npy')
```

#### Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")  
>>> np.genfromtxt("my_file.csv", delimiter=',')  
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

### Data Types

```
>>> np.int64  
>>> np.float32  
>>> np.complex  
>>> np.bool  
>>> np.object  
>>> np.string_  
>>> np.unicode_
```

Signed 64-bit integer types  
Standard double-precision floating point  
Complex numbers represented by 128 floats  
Boolean type storing TRUE and FALSE values  
Python object type  
Fixed-length string type  
Fixed-length unicode type

### Inspecting Your Array

```
>>> a.shape  
>>> len(a)  
>>> b.ndim  
>>> e.size  
>>> b.dtype  
>>> b.dtype.name  
>>> b.astype(int)
```

Array dimensions  
Length of array  
Number of array dimensions  
Number of array elements  
Data type of array elements  
Name of data type  
Convert an array to a different type

### Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

### Array Mathematics

#### Arithmetic Operations

```
>>> g = a - b  
array([[ -0.5,  0. ,  0. ],  
       [ -3. , -3. , -3. ]])  
>>> np.subtract(a,b)  
>>> b + a  
array([[ 2.5,  4. ,  6. ],  
       [ 5. ,  7. ,  9. ]])  
>>> np.add(b,a)  
>>> a / b  
array([[ 0.66666667,  1. ,  1. ],  
       [ 0.25 ,  0.4 ,  0.5 ]])  
>>> np.divide(a,b)  
>>> a * b  
array([[ 1.5,  4. ,  9. ],  
       [ 4. , 10. , 18. ]])  
>>> np.multiply(a,b)  
>>> np.exp(b)  
>>> np.sqrt(b)  
>>> np.sin(a)  
>>> np.cos(b)  
>>> np.log(a)  
>>> e.dot(f)  
array([[ 7. ,  7. ],  
       [ 7. ,  7.]])
```

Subtraction  
Subtraction  
Addition  
Addition  
Division  
Division  
Multiplication  
Multiplication  
Exponentiation  
Square root  
Print sines of an array  
Element-wise cosine  
Element-wise natural logarithm  
Dot product

#### Comparison

```
>>> a == b  
array([[False,  True,  True],  
       [False, False, False]], dtype=bool)  
>>> a < 2  
array([[True, False, False], dtype=bool)  
>>> np.array_equal(a, b)
```

Element-wise comparison  
Element-wise comparison  
Array-wise comparison

#### Aggregate Functions

```
>>> a.sum()  
>>> a.min()  
>>> b.max(axis=0)  
>>> b.cumsum(axis=1)  
>>> a.mean()  
>>> b.median()  
>>> a.corrcoef()  
>>> np.std(b)
```

Array-wise sum  
Array-wise minimum value  
Maximum value of an array row  
Cumulative sum of the elements  
Mean  
Median  
Correlation coefficient  
Standard deviation

### Copying Arrays

```
>>> h = a.view()  
>>> np.copy(a)  
>>> h = a.copy()
```

Create a view of the array with the same data  
Create a copy of the array  
Create a deep copy of the array

### Sorting Arrays

```
>>> a.sort()  
>>> c.sort(axis=0)
```

Sort an array  
Sort the elements of an array's axis

### Subsetting, Slicing, Indexing

Also see Lists

#### Subsetting

```
>>> a[2]  
3  
>>> b[1,2]  
6.0
```

Select the element at the 2nd index  
Select the element at row 1 column 2 (equivalent to b[1][2])

#### Slicing

```
>>> a[0:2]  
array([1, 2])  
>>> b[0:2,1]  
array([ 2.,  5.])
```

Select items at index 0 and 1  
Select items at rows 0 and 1 in column 1

```
>>> b[:1]  
array([[1.5, 2., 3.]])  
>>> c[1,...]  
array([[ 3.,  2.,  1.],  
       [ 4.,  5.,  6.]])
```

Select all items at row 0 (equivalent to b[0:1, :])  
Same as [1, :, :]

```
>>> a[: :-1]  
array([3, 2, 1])
```

Reversed array a

#### Boolean Indexing

```
>>> a[a<2]  
array([1])
```

Select elements from a less than 2

#### Fancy Indexing

```
>>> b[[1, 0, 1, 0], [0, 1, 2, 0]]  
array([ 4. ,  2. ,  6. , 1.5])  
>>> b[[1, 0, 1, 0]][:[0,1,2,0]]  
array([[ 4. ,  5. ,  6. ,  4. ],  
       [ 1.5,  2. ,  3. , 1.5],  
       [ 4. ,  5. ,  6. ,  4. ],  
       [ 1.5,  2. ,  3. , 1.5]])
```

Select elements (1,0), (0,1), (1,2) and (0,0)  
Select a subset of the matrix's rows and columns

### Array Manipulation

#### Transposing Array

```
>>> i = np.transpose(b)  
>>> i.T
```

Permute array dimensions  
Permute array dimensions

#### Changing Array Shape

```
>>> b.ravel()  
>>> g.reshape(3,-2)
```

Flatten the array  
Reshape, but don't change data

#### Adding/Removing Elements

```
>>> h.resize((2,6))  
>>> np.append(h,g)  
>>> np.insert(a, 1, 5)  
>>> np.delete(a, [1])
```

Return a new array with shape (2,6)  
Append items to an array  
Insert items in an array  
Delete items from an array

#### Combining Arrays

```
>>> np.concatenate((a,d),axis=0)  
array([ 1,  2,  3, 10, 15, 20])  
>>> np.vstack((a,b))  
array([[ 1. ,  2. ,  3. ],  
       [ 1.5,  2. ,  3. ],  
       [ 4. ,  5. ,  6. ]])  
>>> np.r_[e,f]  
>>> np.hstack((e,f))  
array([[ 7.,  7.,  1.,  0.],  
       [ 7.,  7.,  0.,  1.]])  
>>> np.column_stack((a,d))  
array([[ 1, 10],  
       [ 2, 15],  
       [ 3, 20]])  
>>> np.c_[a,d]
```

Concatenate arrays  
Stack arrays vertically (row-wise)  
Stack arrays vertically (row-wise)  
Stack arrays horizontally (column-wise)  
Create stacked column-wise arrays  
Create stacked column-wise arrays

#### Splitting Arrays

```
>>> np.hsplit(a,3)  
[array([1]),array([2]),array([3])]  
>>> np.vsplit(c,2)  
[array([[ 1.5,  2. ,  1. ],  
       [ 4. ,  5. ,  6. ]]),  
 array([[ 3.,  2.,  3.],  
       [ 4. ,  5. ,  6.]])]
```

Split the array horizontally at the 3rd index  
Split the array vertically at the 2nd index

DataCamp

Learn Python for Data Science Interactively



# Python For Data Science Cheat Sheet

## Pandas Basics

Learn Python for Data Science Interactively at [www.DataCamp.com](https://www.datacamp.com)



### Pandas

The Pandas library is built on NumPy and provides easy-to-use data structures and data analysis tools for the Python programming language.



Use the following import convention:

```
>>> import pandas as pd
```

### Pandas Data Structures

#### Series

A one-dimensional labeled array capable of holding any data type

a	3
b	-5
c	7
d	4

Index

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

#### DataFrame

	Country	Capital	Population
0	Belgium	Brussels	11190846
1	India	New Delhi	1303171035
2	Brazil	Brasília	207847528

A two-dimensional labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasília'],
            'Population': [11190846, 1303171035, 207847528]}
```

```
>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

### I/O

#### Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

#### Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')

Read multiple sheets from the same file
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

### Asking For Help

```
>>> help(pd.Series.loc)
```

### Selection

Also see NumPy Arrays

#### Getting

```
>>> s['b']
-5

>>> df[1:]
   Country  Capital  Population
1   India  New Delhi  1303171035
2  Brazil  Brasília  207847528
```

Get one element

Get subset of a DataFrame

#### Selecting, Boolean Indexing & Setting

##### By Position

```
>>> df.iloc[[0], [0]]
'Belgium'

>>> df.iat([0], [0])
'Belgium'
```

Select single value by row & column

##### By Label

```
>>> df.loc[[0], ['Country']]
'Belgium'

>>> df.at([0], ['Country'])
'Belgium'
```

Select single value by row & column labels

##### By Label/Position

```
>>> df.ix[2]
Country      Brazil
Capital      Brasília
Population    207847528
```

Select single row of subset of rows

```
>>> df.ix[:, 'Capital']
0      Brussels
1    New Delhi
2     Brasilia
```

Select a single column of subset of columns

```
>>> df.ix[1, 'Capital']
'New Delhi'
```

Select rows and columns

##### Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population'] > 1200000000]
```

Series s where value is not >1  
s where value is <-1 or >2  
Use filter to adjust DataFrame

##### Setting

```
>>> s['a'] = 6
```

Set index a of Series s to 6

### Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)  
Drop values from columns(axis=1)

### Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis  
Sort by the values along an axis  
Assign ranks to entries

### Retrieving Series/DataFrame Information

#### Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows,columns)  
Describe index  
Describe DataFrame columns  
Info on DataFrame  
Number of non-NA values

#### Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idxmin()/df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values  
Cumulative sum of values  
Minimum/maximum values  
Minimum/Maximum index value  
Summary statistics  
Mean of values  
Median of values

### Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function  
Apply function element-wise

### Data Alignment

#### Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
a      10.0
b       NaN
c       5.0
d       7.0
```

#### Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
a      10.0
b     -5.0
c       5.0
d       7.0

>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

DataCamp

Learn Python for Data Science Interactively





# Python For Data Science Cheat Sheet

## Pandas

Learn Python for Data Science Interactively at [www.DataCamp.com](https://www.datacamp.com)



### Reshaping Data

#### Pivot

```
>>> df3= df2.pivot(index='Date',
                    columns='Type',
                    values='Value')
```

Spread rows into columns

	Date	Type	Value
0	2016-03-01	a	11.432
1	2016-03-02	b	13.031
2	2016-03-01	c	20.784
3	2016-03-03	a	99.906
4	2016-03-02	a	1.303
5	2016-03-03	c	20.784

	Type	a	b	c
Date				
2016-03-01		11.432	NaN	20.784
2016-03-02		1.303	13.031	NaN
2016-03-03		99.906	NaN	20.784

#### Pivot Table

```
>>> df4 = pd.pivot_table(df2,
                        values='Value',
                        index='Date',
                        columns='Type')
```

Spread rows into columns

#### Stack / Unstack

```
>>> stacked = df5.stack()
>>> stacked.unstack()
```

Pivot a level of column labels  
Pivot a level of index labels

		0	1
1	5	0.233482	0.390959
2	4	0.184713	0.237102
3	3	0.433522	0.429401

Unstacked

			0	1	2
1	5	0	0.233482		
1	5	1	0.390959		
2	4	0	0.184713		
2	4	1	0.237102		
3	3	0	0.433522		
3	3	1	0.429401		

Stacked

#### Melt

```
>>> pd.melt(df2,
            id_vars=["Date"],
            value_vars=["Type", "Value"],
            value_name="Observations")
```

Gather columns into rows

	Date	Type	Value
0	2016-03-01	a	11.432
1	2016-03-02	b	13.031
2	2016-03-01	c	20.784
3	2016-03-03	a	99.906
4	2016-03-02	a	1.303
5	2016-03-03	c	20.784

	Date	Variable	Observations
0	2016-03-01	Type	a
1	2016-03-02	Type	b
2	2016-03-01	Type	c
3	2016-03-03	Type	a
4	2016-03-02	Type	a
5	2016-03-03	Type	c
6	2016-03-01	Value	11.432
7	2016-03-02	Value	13.031
8	2016-03-01	Value	20.784
9	2016-03-03	Value	99.906
10	2016-03-02	Value	1.303
11	2016-03-03	Value	20.784

### Iteration

```
>>> df.iteritems()
>>> df.iterrows()
```

(Column-index, Series) pairs  
(Row-index, Series) pairs

### Advanced Indexing

Also see NumPy Arrays

#### Selecting

```
>>> df3.loc[:, (df3>1).any()]
>>> df3.loc[:, (df3>1).all()]
>>> df3.loc[:, df3.isnull().any()]
>>> df3.loc[:, df3.notnull().all()]
```

Select cols with any vals >1  
Select cols with vals > 1  
Select cols with NaN  
Select cols without NaN

#### Indexing With isin

```
>>> df[(df.Country.isin(df2.Type))]
>>> df3.filter(items=["a", "b"])
>>> df.select(lambda x: not x%5)
```

Find same elements  
Filter on values  
Select specific elements

#### Where

```
>>> s.where(s > 0)
```

Subset the data

#### Query

```
>>> df6.query('second > first')
```

Query DataFrame

### Setting/Resetting Index

```
>>> df.set_index('Country')
>>> df4 = df.reset_index()
>>> df = df.rename(index=str,
                  columns={"Country": "entry",
                           "Capital": "cptl",
                           "Population": "ppltn"})
```

Set the index  
Reset the index  
Rename DataFrame

### Reindexing

```
>>> s2 = s.reindex(['a', 'c', 'd', 'e', 'b'])
```

#### Forward Filling

```
>>> df.reindex(range(4),
               method='ffill')
   Country Capital Population
0  Belgium Brussels  1190846
1   India New Delhi  1303171035
2  Brazil Brasilia  207847528
3  Brazil Brasilia  207847528
```

#### Backward Filling

```
>>> s3 = s.reindex(range(5),
                   method='bfill')
0  3
1  3
2  3
3  3
4  3
```

### MultiIndexing

```
>>> arrays = [np.array([1,2,3]),
              np.array([5,4,3])]
>>> df5 = pd.DataFrame(np.random.rand(3, 2), index=arrays)
>>> tuples = list(zip(*arrays))
>>> index = pd.MultiIndex.from_tuples(tuples,
                                     names=['first', 'second'])
>>> df6 = pd.DataFrame(np.random.rand(3, 2), index=index)
>>> df2.set_index(["Date", "Type"])
```

### Duplicate Data

```
>>> s3.unique()
>>> df2.duplicated('Type')
>>> df2.drop_duplicates('Type', keep='last')
>>> df.index.duplicated()
```

Return unique values  
Check duplicates  
Drop duplicates  
Check index duplicates

### Grouping Data

#### Aggregation

```
>>> df2.groupby(by=['Date', 'Type']).mean()
>>> df4.groupby(level=0).sum()
>>> df4.groupby(level=0).agg({'a': lambda x: sum(x)/len(x),
                           'b': np.sum})
```

#### Transformation

```
>>> customSum = lambda x: (x+x%2)
>>> df4.groupby(level=0).transform(customSum)
```

### Missing Data

```
>>> df.dropna()
>>> df3.fillna(df3.mean())
>>> df2.replace("a", "f")
```

Drop NaN values  
Fill NaN values with a predetermined value  
Replace values with others

### Combining Data

data1		data2	
X1	X2	X1	X3
a	11.432	a	20.784
b	1.303	b	NaN
c	99.906	d	20.784

#### Merge

```
>>> pd.merge(data1,
             data2,
             how='left',
             on='X1')
```

X1	X2	X3
a	11.432	20.784
b	1.303	NaN
c	99.906	NaN

```
>>> pd.merge(data1,
             data2,
             how='right',
             on='X1')
```

X1	X2	X3
a	11.432	20.784
b	1.303	NaN
d	NaN	20.784

```
>>> pd.merge(data1,
             data2,
             how='inner',
             on='X1')
```

X1	X2	X3
a	11.432	20.784
b	1.303	NaN

```
>>> pd.merge(data1,
             data2,
             how='outer',
             on='X1')
```

X1	X2	X3
a	11.432	20.784
b	1.303	NaN
c	99.906	NaN
d	NaN	20.784

#### Join

```
>>> data1.join(data2, how='right')
```

#### Concatenate

##### Vertical

```
>>> s.append(s2)
```

##### Horizontal/Vertical

```
>>> pd.concat([s,s2],axis=1, keys=['One', 'Two'])
>>> pd.concat([data1, data2], axis=1, join='inner')
```

### Dates

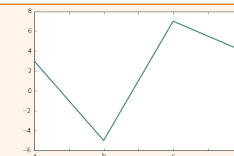
```
>>> df2['Date'] = pd.to_datetime(df2['Date'])
>>> df2['Date'] = pd.date_range('2000-1-1',
                              periods=6,
                              freq='M')
>>> dates = [datetime(2012,5,1), datetime(2012,5,2)]
>>> index = pd.DatetimeIndex(dates)
>>> index = pd.date_range(datetime(2012,2,1), end, freq='BM')
```

### Visualization

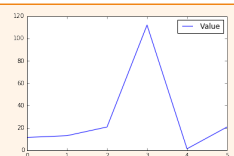
Also see Matplotlib

```
>>> import matplotlib.pyplot as plt
```

```
>>> s.plot()
>>> plt.show()
```



```
>>> df2.plot()
>>> plt.show()
```



DataCamp

Learn Python for Data Science Interactively



# Python For Data Science Cheat Sheet

## Matplotlib

Learn Python Interactively at [www.DataCamp.com](https://www.datacamp.com)



### Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.



### 1 Prepare The Data

Also see Lists & NumPy

#### 1D Data

```
>>> import numpy as np
>>> x = np.linspace(0, 10, 100)
>>> y = np.cos(x)
>>> z = np.sin(x)
```

#### 2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))
>>> data2 = 3 * np.random.random((10, 10))
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]
>>> U = -1 - X**2 + Y
>>> V = 1 + X - Y**2
>>> from matplotlib.cbook import get_sample_data
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

### 2 Create Plot

```
>>> import matplotlib.pyplot as plt
```

#### Figure

```
>>> fig = plt.figure()
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

#### Axes

All plotting is done with respect to an Axes. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()
>>> ax1 = fig.add_subplot(221) # row-col-num
>>> ax3 = fig.add_subplot(212)
>>> fig3, axes = plt.subplots(nrows=2, ncols=2)
>>> fig4, axes2 = plt.subplots(ncols=3)
```

### 3 Plotting Routines

#### 1D Data

```
>>> fig, ax = plt.subplots()
>>> lines = ax.plot(x,y)
>>> ax.scatter(x,y)
>>> axes[0,0].bar([1,2,3],[3,4,5])
>>> axes[1,0].barh([0.5,1,2.5],[0,1,2])
>>> axes[1,1].axhline(0.45)
>>> axes[0,1].axvline(0.65)
>>> ax.fill(x,y,color='blue')
>>> ax.fill_between(x,y,color='yellow')
```

Draw points with lines or markers connecting them  
Draw unconnected points, scaled or colored  
Plot vertical rectangles (constant width)  
Plot horizontal rectangles (constant height)  
Draw a horizontal line across axes  
Draw a vertical line across axes  
Draw filled polygons  
Fill between y-values and 0

#### 2D Data or Images

```
>>> fig, ax = plt.subplots()
>>> im = ax.imshow(img,
                  cmap='gist_earth',
                  interpolation='nearest',
                  vmin=-2,
                  vmax=2)
```

Colormapped or RGB arrays

#### Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)
>>> axes[1,1].quiver(y,z)
>>> axes[0,1].streamplot(X,Y,U,V)
```

Add an arrow to the axes  
Plot a 2D field of arrows  
Plot a 2D field of arrows

#### Data Distributions

```
>>> ax1.hist(y)
>>> ax3.boxplot(y)
>>> ax3.violinplot(z)
```

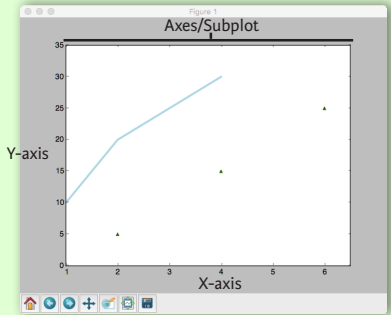
Plot a histogram  
Make a box and whisker plot  
Make a violin plot

```
>>> axes2[0].pcolor(data2)
>>> axes2[0].pcolormesh(data)
>>> CS = plt.contour(Y,X,U)
>>> axes2[2].contourf(data1)
>>> axes2[2] = ax.clabel(CS)
```

Pseudocolor plot of 2D array  
Pseudocolor plot of 2D array  
Plot contours  
Plot filled contours  
Label a contour plot

### Plot Anatomy & Workflow

#### Plot Anatomy



#### Workflow

The basic steps to creating plots with matplotlib are:

- 1 Prepare data
- 2 Create plot
- 3 Plot
- 4 Customize plot
- 5 Save plot
- 6 Show plot

```
>>> import matplotlib.pyplot as plt
>>> x = [1,2,3,4]
>>> y = [10,20,25,30]
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, y, color='lightblue', linewidth=3)
>>> ax.scatter([2,4,6],
              [5,15,25],
              color='darkgreen',
              marker='^')
>>> ax.set_xlim(1, 6.5)
>>> plt.savefig('foo.png')
>>> plt.show()
```

### 4 Customize Plot

#### Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)
>>> ax.plot(x, y, alpha = 0.4)
>>> ax.plot(x, y, c='k')
>>> fig.colorbar(im, orientation='horizontal')
>>> im = ax.imshow(img,
                  cmap='seismic')
```

#### Markers

```
>>> fig, ax = plt.subplots()
>>> ax.scatter(x,y,marker=".")
>>> ax.plot(x,y,marker="o")
```

#### Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)
>>> plt.plot(x,y,ls='solid')
>>> plt.plot(x,y,ls='--')
>>> plt.plot(x,y,'--',x**2,y**2,'-.')
>>> plt.setp(lines,color='r',linewidth=4.0)
```

#### Text & Annotations

```
>>> ax.text(1,
          -2.1,
          'Example Graph',
          style='italic')
>>> ax.annotate("Sine",
              xy=(8, 0),
              xycoords='data',
              xytext=(10.5, 0),
              textcoords='data',
              arrowprops=dict(arrowstyle="->",
                             connectionstyle="arc3"),)
```

#### Mathtext

```
>>> plt.title(r'$\sigma_i=15$', fontsize=20)
```

#### Limits, Legends & Layouts

##### Limits & Autoscaling

```
>>> ax.margins(x=0.0,y=0.1)
>>> ax.axis('equal')
>>> ax.set(xlim=[0,10.5],ylim=[-1.5,1.5])
>>> ax.set_xlim(0,10.5)
```

##### Legends

```
>>> ax.set(title='An Example Axes',
          ylabel='Y-Axis',
          xlabel='X-Axis')
>>> ax.legend(loc='best')
```

##### Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),
               ticklabels=[3,100,-12,"foo"])
>>> ax.tick_params(axis='y',
                  direction='inout',
                  length=10)
```

##### Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,
                        hspace=0.3,
                        left=0.125,
                        right=0.9,
                        top=0.9,
                        bottom=0.1)
```

```
>>> fig.tight_layout()
```

##### Axis Spines

```
>>> ax1.spines['top'].set_visible(False)
>>> ax1.spines['bottom'].set_position(('outward',10))
```

Add padding to a plot  
Set the aspect ratio of the plot to 1  
Set limits for x-and y-axis  
Set limits for x-axis

Set a title and x-and y-axis labels

No overlapping plot elements

Manually set x-ticks

Make y-ticks longer and go in and out

Adjust the spacing between subplots

Fit subplot(s) in to the figure area

Make the top axis line for a plot invisible  
Move the bottom axis line outward

### 5 Save Plot

#### Save figures

```
>>> plt.savefig('foo.png')
```

#### Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

### 6 Show Plot

```
>>> plt.show()
```

### Close & Clear

```
>>> plt.cla()
>>> plt.clf()
>>> plt.close()
```

Clear an axis  
Clear the entire figure  
Close a window



# Python For Data Science Cheat Sheet

## Scikit-Learn

Learn Python for data science [Interactively](#) at [www.DataCamp.com](#)



### Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.



#### A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

#### Loading The Data

Also see NumPy & Pandas

Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.

```
>>> import numpy as np
>>> X = np.random.random((10,5))
>>> y = np.array(['M', 'M', 'F', 'F', 'M', 'F', 'M', 'F', 'F', 'F'])
>>> X[X < 0.7] = 0
```

#### Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    random_state=0)
```

### Preprocessing The Data

#### Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

#### Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

#### Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

### Create Your Model

#### Supervised Learning Estimators

```
Linear Regression
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)

Support Vector Machines (SVM)
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')

Naive Bayes
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()

KNN
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

#### Unsupervised Learning Estimators

```
Principal Component Analysis (PCA)
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)

K Means
>>> from sklearn.cluster import KMeans
>>> k_means = KMeans(n_clusters=3, random_state=0)
```

### Model Fitting

<b>Supervised learning</b>	Fit the model to the data
<pre>&gt;&gt;&gt; lr.fit(X, y) &gt;&gt;&gt; knn.fit(X_train, y_train) &gt;&gt;&gt; svc.fit(X_train, y_train)</pre>	
<b>Unsupervised Learning</b>	Fit the model to the data
<pre>&gt;&gt;&gt; k_means.fit(X_train) &gt;&gt;&gt; pca_model = pca.fit_transform(X_train)</pre>	Fit to data, then transform it

### Prediction

<b>Supervised Estimators</b>	Predict labels
<pre>&gt;&gt;&gt; y_pred = svc.predict(np.random.random((2,5))) &gt;&gt;&gt; y_pred = lr.predict(X_test) &gt;&gt;&gt; y_pred = knn.predict_proba(X_test)</pre>	Predict labels
<b>Unsupervised Estimators</b>	Estimate probability of a label
<pre>&gt;&gt;&gt; y_pred = k_means.predict(X_test)</pre>	Predict labels in clustering algos

### Evaluate Your Model's Performance

#### Classification Metrics

<b>Accuracy Score</b>	Estimator score method
<pre>&gt;&gt;&gt; knn.score(X_test, y_test) &gt;&gt;&gt; from sklearn.metrics import accuracy_score &gt;&gt;&gt; accuracy_score(y_test, y_pred)</pre>	Metric scoring functions
<b>Classification Report</b>	Precision, recall, f1-score and support
<pre>&gt;&gt;&gt; from sklearn.metrics import classification_report &gt;&gt;&gt; print(classification_report(y_test, y_pred))</pre>	
<b>Confusion Matrix</b>	
<pre>&gt;&gt;&gt; from sklearn.metrics import confusion_matrix &gt;&gt;&gt; print(confusion_matrix(y_test, y_pred))</pre>	

#### Regression Metrics

<b>Mean Absolute Error</b>
<pre>&gt;&gt;&gt; from sklearn.metrics import mean_absolute_error &gt;&gt;&gt; y_true = [3, -0.5, 2] &gt;&gt;&gt; mean_absolute_error(y_true, y_pred)</pre>
<b>Mean Squared Error</b>
<pre>&gt;&gt;&gt; from sklearn.metrics import mean_squared_error &gt;&gt;&gt; mean_squared_error(y_true, y_pred)</pre>
<b>R<sup>2</sup> Score</b>
<pre>&gt;&gt;&gt; from sklearn.metrics import r2_score &gt;&gt;&gt; r2_score(y_true, y_pred)</pre>

#### Clustering Metrics

<b>Adjusted Rand Index</b>
<pre>&gt;&gt;&gt; from sklearn.metrics import adjusted_rand_score &gt;&gt;&gt; adjusted_rand_score(y_true, y_pred)</pre>
<b>Homogeneity</b>
<pre>&gt;&gt;&gt; from sklearn.metrics import homogeneity_score &gt;&gt;&gt; homogeneity_score(y_true, y_pred)</pre>
<b>V-measure</b>
<pre>&gt;&gt;&gt; from sklearn.metrics import v_measure_score &gt;&gt;&gt; metrics.v_measure_score(y_true, y_pred)</pre>

#### Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

### Tune Your Model

#### Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,5),
            "metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
                      param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

#### Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5),
            "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knn,
                               param_distributions=params,
                               cv=4,
                               n_iter=8,
                               random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```



# Beginner's Python Cheat Sheet

## Variables and Strings

*Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.*

### Hello world

```
print("Hello world!")
```

### Hello world with a variable

```
msg = "Hello world!"  
print(msg)
```

### Concatenation (combining strings)

```
first_name = 'albert'  
last_name = 'einstein'  
full_name = first_name + ' ' + last_name  
print(full_name)
```

## Lists

*A list stores a series of items in a particular order. You access items using an index, or within a loop.*

### Make a list

```
bikes = ['trek', 'redline', 'giant']
```

### Get the first item in a list

```
first_bike = bikes[0]
```

### Get the last item in a list

```
last_bike = bikes[-1]
```

### Looping through a list

```
for bike in bikes:  
    print(bike)
```

### Adding items to a list

```
bikes = []  
bikes.append('trek')  
bikes.append('redline')  
bikes.append('giant')
```

### Making numerical lists

```
squares = []  
for x in range(1, 11):  
    squares.append(x**2)
```

## Lists (cont.)

### List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

### Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']  
first_two = finishers[:2]
```

### Copying a list

```
copy_of_bikes = bikes[:]
```

## Tuples

*Tuples are similar to lists, but the items in a tuple can't be modified.*

### Making a tuple

```
dimensions = (1920, 1080)
```

## If statements

*If statements are used to test for particular conditions and respond appropriately.*

### Conditional tests

equals	x == 42
not equal	x != 42
greater than	x > 42
or equal to	x >= 42
less than	x < 42
or equal to	x <= 42

### Conditional test with lists

```
'trek' in bikes  
'surly' not in bikes
```

### Assigning boolean values

```
game_active = True  
can_edit = False
```

### A simple if test

```
if age >= 18:  
    print("You can vote!")
```

### If-elif-else statements

```
if age < 4:  
    ticket_price = 0  
elif age < 18:  
    ticket_price = 10  
else:  
    ticket_price = 15
```

## Dictionaries

*Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.*

### A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

### Accessing a value

```
print("The alien's color is " + alien['color'])
```

### Adding a new key-value pair

```
alien['x_position'] = 0
```

### Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}  
for name, number in fav_numbers.items():  
    print(name + ' loves ' + str(number))
```

### Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}  
for name in fav_numbers.keys():  
    print(name + ' loves a number')
```

### Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}  
for number in fav_numbers.values():  
    print(str(number) + ' is a favorite')
```

## User input

*Your programs can prompt the user for input. All input is stored as a string.*

### Prompting for a value

```
name = input("What's your name? ")  
print("Hello, " + name + "!")
```

### Prompting for numerical input

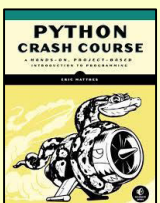
```
age = input("How old are you? ")  
age = int(age)
```

```
pi = input("What's the value of pi? ")  
pi = float(pi)
```

## Python Crash Course

*Covers Python 3 and Python 2*

[nostarchpress.com/pythoncrashcourse](http://nostarchpress.com/pythoncrashcourse)





## While loops

A while loop repeats a block of code as long as a certain condition is true.

### A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

### Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

## Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

### A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")
```

```
greet_user()
```

### Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")
```

```
greet_user('jesse')
```

### Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")
```

```
make_pizza()
make_pizza('pepperoni')
```

### Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y
```

```
sum = add_numbers(3, 5)
print(sum)
```

## Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

### Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")
```

```
my_dog = Dog('Peso')
```

```
print(my_dog.name + " is a great dog!")
my_dog.sit()
```

### Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")
```

```
my_dog = SARDog('Willie')
```

```
print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

## Infinite Skills

*If you had infinite programming skills, what would you build?*

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

## Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

### Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

### Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

### Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

## Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

### Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

## Zen of Python

*Simple is better than complex*

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

*More cheat sheets available at*  
[ehmatthes.github.io/pcc/](https://ehmatthes.github.io/pcc/)

# Beginner's Python Cheat Sheet - Lists

## What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

## Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

## Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

## Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

## Getting the first element

```
first_user = users[0]
```

## Getting the second element

```
second_user = users[1]
```

## Getting the last element

```
newest_user = users[-1]
```

## Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

## Changing an element

```
users[0] = 'valerie'  
users[-2] = 'ronald'
```

## Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

## Adding an element to the end of the list

```
users.append('amy')
```

## Starting with an empty list

```
users = []  
users.append('val')  
users.append('bob')  
users.append('mia')
```

## Inserting elements at a particular position

```
users.insert(0, 'joe')  
users.insert(3, 'bea')
```

## Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

## Deleting an element by its position

```
del users[-1]
```

## Removing an item by its value

```
users.remove('mia')
```

## Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

## Pop the last item from a list

```
most_recent_user = users.pop()  
print(most_recent_user)
```

## Pop the first item in a list

```
first_user = users.pop(0)  
print(first_user)
```

## List length

The len() function returns the number of items in a list.

## Find the length of a list

```
num_users = len(users)  
print("We have " + str(num_users) + " users.")
```

## Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

## Sorting a list permanently

```
users.sort()
```

## Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

## Sorting a list temporarily

```
print(sorted(users))  
print(sorted(users, reverse=True))
```

## Reversing the order of a list

```
users.reverse()
```

## Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

## Printing all items in a list

```
for user in users:  
    print(user)
```

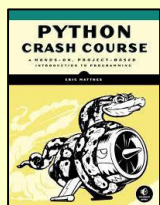
## Printing a message for each item, and a separate message afterwards

```
for user in users:  
    print("Welcome, " + user + "!")  
  
print("Welcome, we're glad to see you all!")
```

## Python Crash Course

Covers Python 3 and Python 2

[nostarchpress.com/pythoncrashcourse](http://nostarchpress.com/pythoncrashcourse)



## The range() function

You can use the range() function to work with a set of numbers efficiently. The range() function starts at 0 by default, and stops one number below the number passed to it. You can use the list() function to efficiently generate a large list of numbers.

### Printing the numbers 0 to 1000

```
for number in range(1001):  
    print(number)
```

### Printing the numbers 1 to 1000

```
for number in range(1, 1001):  
    print(number)
```

### Making a list of numbers from 1 to a million

```
numbers = list(range(1, 1000001))
```

## Simple statistics

There are a number of simple statistics you can run on a list containing numerical data.

### Finding the minimum value in a list

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]  
youngest = min(ages)
```

### Finding the maximum value

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]  
oldest = max(ages)
```

### Finding the sum of all values

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]  
total_years = sum(ages)
```

## Slicing a list

You can work with any set of elements from a list. A portion of a list is called a slice. To slice a list start with the index of the first item you want, then add a colon and the index after the last item you want. Leave off the first index to start at the beginning of the list, and leave off the last index to slice through the end of the list.

### Getting the first three items

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']  
first_three = finishers[:3]
```

### Getting the middle three items

```
middle_three = finishers[1:4]
```

### Getting the last three items

```
last_three = finishers[-3:]
```

## Copying a list

To copy a list make a slice that starts at the first item and ends at the last item. If you try to copy a list without using this approach, whatever you do to the copied list will affect the original list as well.

### Making a copy of a list

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']  
copy_of_finishers = finishers[:]
```

## List comprehensions

You can use a loop to generate a list based on a range of numbers or on another list. This is a common operation, so Python offers a more efficient way to do it. List comprehensions may look complicated at first; if so, use the for loop approach until you're ready to start using comprehensions.

To write a comprehension, define an expression for the values you want to store in the list. Then write a for loop to generate input values needed to make the list.

### Using a loop to generate a list of square numbers

```
squares = []  
for x in range(1, 11):  
    square = x**2  
    squares.append(square)
```

### Using a comprehension to generate a list of square numbers

```
squares = [x**2 for x in range(1, 11)]
```

### Using a loop to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']  
  
upper_names = []  
for name in names:  
    upper_names.append(name.upper())
```

### Using a comprehension to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']  
  
upper_names = [name.upper() for name in names]
```

## Styling your code

### Readability counts

- Use four spaces per indentation level.
- Keep your lines to 79 characters or fewer.
- Use single blank lines to group parts of your program visually.

## Tuples

A tuple is like a list, except you can't change the values in a tuple once it's defined. Tuples are good for storing information that shouldn't be changed throughout the life of a program. Tuples are designated by parentheses instead of square brackets. (You can overwrite an entire tuple, but you can't change the individual elements in a tuple.)

### Defining a tuple

```
dimensions = (800, 600)
```

### Looping through a tuple

```
for dimension in dimensions:  
    print(dimension)
```

### Overwriting a tuple

```
dimensions = (800, 600)  
print(dimensions)
```

```
dimensions = (1200, 900)
```

## Visualizing your code

When you're first learning about data structures such as lists, it helps to visualize how Python is working with the information in your program. [pythontutor.com](http://pythontutor.com) is a great tool for seeing how Python keeps track of the information in a list. Try running the following code on [pythontutor.com](http://pythontutor.com), and then run your own code.

### Build a list and print the items in the list

```
dogs = []  
dogs.append('willie')  
dogs.append('hootz')  
dogs.append('peso')  
dogs.append('goblin')
```

```
for dog in dogs:  
    print("Hello " + dog + "!")  
print("I love these dogs!")
```

```
print("\nThese were my first two dogs:")  
old_dogs = dogs[:2]  
for old_dog in old_dogs:  
    print(old_dog)
```

```
del dogs[0]  
dogs.remove('peso')  
print(dogs)
```

More cheat sheets available at  
[ehmatthes.github.io/pcc/](https://ehmatthes.github.io/pcc/)

# Beginner's Python Cheat Sheet — Dictionaries

## What are dictionaries?

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.

## Defining a dictionary

*Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.*

### Making a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
```

## Accessing values

*To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.*

*You can also use the `get()` method, which returns `None` instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.*

### Getting the value associated with a key

```
alien_0 = {'color': 'green', 'points': 5}
```

```
print(alien_0['color'])  
print(alien_0['points'])
```

### Getting the value with `get()`

```
alien_0 = {'color': 'green'}
```

```
alien_color = alien_0.get('color')  
alien_points = alien_0.get('points', 0)
```

```
print(alien_color)  
print(alien_points)
```

## Adding new key-value pairs

*You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.*

*This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.*

### Adding a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}
```

```
alien_0['x'] = 0  
alien_0['y'] = 25  
alien_0['speed'] = 1.5
```

### Adding to an empty dictionary

```
alien_0 = {}  
alien_0['color'] = 'green'  
alien_0['points'] = 5
```

## Modifying values

*You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.*

### Modifying values in a dictionary

```
alien_0 = {'color': 'green', 'points': 5}  
print(alien_0)
```

```
# Change the alien's color and point value.  
alien_0['color'] = 'yellow'  
alien_0['points'] = 10  
print(alien_0)
```

## Removing key-value pairs

*You can remove any key-value pair you want from a dictionary. To do so use the `del` keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.*

### Deleting a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}  
print(alien_0)
```

```
del alien_0['points']  
print(alien_0)
```

## Visualizing dictionaries

*Try running some of these examples on [pythontutor.com](http://pythontutor.com).*

## Looping through a dictionary

*You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.*

*A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.*

### Looping through all key-value pairs

```
# Store people's favorite languages.  
fav_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'ruby',  
    'phil': 'python',  
}
```

```
# Show each person's favorite language.  
for name, language in fav_languages.items():  
    print(name + ": " + language)
```

### Looping through all the keys

```
# Show everyone who's taken the survey.  
for name in fav_languages.keys():  
    print(name)
```

### Looping through all the values

```
# Show all the languages that have been chosen.  
for language in fav_languages.values():  
    print(language)
```

### Looping through all the keys in order

```
# Show each person's favorite language,  
# in order by the person's name.  
for name in sorted(fav_languages.keys()):  
    print(name + ": " + language)
```

## Dictionary length

*You can find the number of key-value pairs in a dictionary.*

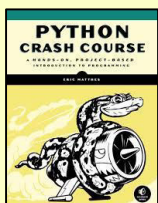
### Finding a dictionary's length

```
num_responses = len(fav_languages)
```

## Python Crash Course

*Covers Python 3 and Python 2*

[nostarchpress.com/pythoncrashcourse](http://nostarchpress.com/pythoncrashcourse)





## Nesting — A list of dictionaries

*It's sometimes useful to store a set of dictionaries in a list; this is called nesting.*

### Storing dictionaries in a list

```
# Start with an empty list.
users = []

# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
}
users.append(new_user)

# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
}
users.append(new_user)

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

You can also define a list of dictionaries directly, without using `append()`:

```
# Define a list of users, where each user
# is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

## Nesting — Lists in a dictionary

*Storing a list inside a dictionary allows you to associate more than one value with each key.*

### Storing lists in a dictionary

```
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

## Nesting — A dictionary of dictionaries

*You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.*

### Storing dictionaries in a dictionary

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
    },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
    },
}

for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']

    print("\tFull name: " + full_name.title())
    print("\tLocation: " + location.title())
```

## Levels of nesting

*Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.*

## Using an OrderedDict

*Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an `OrderedDict`.*

### Preserving the order of keys and values

```
from collections import OrderedDict

# Store each person's languages, keeping
# track of who responded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
# were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

## Generating a million dictionaries

*You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.*

### A million aliens

```
aliens = []

# Make a million green aliens, worth 5 points
# each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

*More cheat sheets available at*  
[ehmatthes.github.io/pcc/](https://ehmatthes.github.io/pcc/)

# Beginner's Python Cheat Sheet — If Statements and While Loops

## What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that identify the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

## Conditional Tests

*A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.*

### Checking for equality

*A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.*

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

### Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

### Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

## Numerical comparisons

*Testing numerical values is similar to testing string values.*

### Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

### Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

## Checking multiple conditions

*You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.*

### Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

### Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

## Boolean values

*A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.*

### Simple boolean values

```
game_active = True
can_edit = False
```

## If statements

*Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.*

### Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

### If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

### The if-elif-else chain

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
else:
    price = 10

print("Your cost is $" + str(price) + ".")
```

## Conditional tests with lists

*You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.*

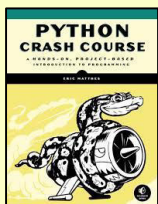
### Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

## Python Crash Course

*Covers Python 3 and Python 2*

[nostarchpress.com/pythoncrashcourse](http://nostarchpress.com/pythoncrashcourse)



## Conditional tests with lists (cont.)

### Testing if a value is not in a list

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'
```

```
if user not in banned_users:
    print("You can play!")
```

### Checking if a list is empty

```
players = []
```

```
if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

## Accepting input

*You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.*

### Simple input

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

### Accepting numerical input

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

### Accepting input in Python 2.7

*Use raw\_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.*

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

## While loops

*A while loop repeats a block of code as long as a condition is True.*

### Counting to 5

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

## While loops (cont.)

### Letting the user choose when to quit

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "
```

```
message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

### Using a flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "
```

```
active = True
while active:
    message = input(prompt)
```

```
    if message == 'quit':
        active = False
    else:
        print(message)
```

### Using break to exit a loop

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "
```

```
while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

## Accepting input with Sublime Text

*Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.*

## Breaking out of loops

*You can use the break statement and the continue statement with any of Python's loops. For example you can use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.*

## While loops (cont.)

### Using continue in a loop

```
banned_users = ['eve', 'fred', 'gary', 'helen']
```

```
prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "
```

```
players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)
```

```
print("\nYour team:")
for player in players:
    print(player)
```

## Avoiding infinite loops

*Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.*

### An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

## Removing all instances of a value from a list

*The remove() method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.*

### Removing all cats from a list of pets

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)
```

```
while 'cat' in pets:
    pets.remove('cat')
```

```
print(pets)
```

*More cheat sheets available at*  
[ehmatthes.github.io/pcc/](https://ehmatthes.github.io/pcc/)

# Beginner's Python Cheat Sheet — Functions

## What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

## Defining a function

*The first line of a function is its definition, marked by the keyword `def`. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.*

*To call a function, give the name of the function followed by a set of parentheses.*

## Making a function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

## Passing information to a function

*Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.*

## Passing a single argument

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

## Positional and keyword arguments

*The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.*

*With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.*

## Using positional arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

## Using keyword arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

## Default values

*You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.*

## Using a default value

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

## Using None to make an argument optional

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

## Return values

*A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.*

## Returning a single value

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

## Returning a dictionary

```
def build_person(first, last):
    """Return a dictionary of information
    about a person."""
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

## Returning a dictionary with optional values

```
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person."""
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

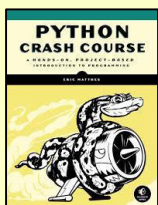
## Visualizing functions

*Try running some of these examples on [pythontutor.com](http://pythontutor.com).*

## Python Crash Course

*Covers Python 3 and Python 2*

[nostarchpress.com/pythoncrashcourse](http://nostarchpress.com/pythoncrashcourse)





## Passing a list to a function

You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.

### Passing a list as an argument

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

### Allowing a function to modify a list

The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
# and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

### Preventing a function from modifying a list

The following example is the same as the previous one, except the original list is unchanged after calling print\_models().

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
# and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

## Passing an arbitrary number of arguments

Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the \* operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition.

The \*\* operator allows a parameter to collect an arbitrary number of keyword arguments.

### Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
            'onions', 'extra cheese')
```

### Collecting an arbitrary number of keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
# of information.
user_0 = build_profile('albert', 'einstein',
                       location='princeton')
user_1 = build_profile('marie', 'curie',
                       location='paris', field='chemistry')

print(user_0)
print(user_1)
```

## What's the best way to structure a function?

As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.

## Modules

You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)

### Storing a function in a module

File: pizza.py

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

### Importing an entire module

File: making\_pizzas.py

Every function in the module is available in the program file.

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

### Importing a specific function

Only the imported functions are available in the program file.

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

### Giving a module an alias

```
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

### Giving a function an alias

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

### Importing all functions from a module

Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.

```
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at  
[ehmatthes.github.io/pcc/](https://ehmatthes.github.io/pcc/)

Seaborn

Learn Data Science Interactively at [www.DataCamp.com](https://www.datacamp.com)



Statistical Data Visualization With Seaborn

The Python visualization library **Seaborn** is based on **matplotlib** and provides a high-level interface for drawing attractive statistical graphics.

Make use of the following aliases to import the libraries:

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
```

The basic steps to creating plots with Seaborn are:

- 1. Prepare some data
- 2. Control figure aesthetics
- 3. Plot with Seaborn
- 4. Further customize your plot

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips")
>>> sns.set_style("whitegrid")
>>> g = sns.lmplot(x="tip", y="total_bill", data=tips, aspect=2)
>>> g = (g.set_axis_labels("Tip", "Total bill (USD)")).set(xlim=(0,10),ylim=(0,100))
>>> plt.title("title")
>>> plt.show(g)
```

1 Data

Also see Lists, NumPy & Pandas

```
>>> import pandas as pd
>>> import numpy as np
>>> uniform_data = np.random.rand(10, 12)
>>> data = pd.DataFrame({'x':np.arange(1,101), 'y':np.random.normal(0,4,100)})
```

Seaborn also offers built-in data sets:

```
>>> titanic = sns.load_dataset("titanic")
>>> iris = sns.load_dataset("iris")
```

2 Figure Aesthetics

Also see Matplotlib

```
>>> f, ax = plt.subplots(figsize=(5,6))
```

Create a figure and one subplot

Seaborn styles

```
>>> sns.set()
>>> sns.set_style("whitegrid")
>>> sns.set_style("ticks", {'xtick.major.size':8, 'ytick.major.size':8})
>>> sns.axes_style("whitegrid")
```

(Re)set the seaborn default Set the matplotlib parameters Set the matplotlib parameters

Return a dict of params or use with with to temporarily set the style

Axis Grids

```
>>> g = sns.FacetGrid(titanic, col="survived", row="sex")
>>> g = g.map(plt.hist, "age")
>>> sns.factorplot(x="pclass", y="survived", hue="sex", data=titanic)
>>> sns.lmplot(x="sepal_width", y="sepal_length", hue="species", data=iris)
```

Subplot grid for plotting conditional relationships

Draw a categorical plot onto a Facetgrid

Plot data and regression model fits across a FacetGrid

Categorical Plots

```
>>> sns.stripplot(x="species", y="petal_length", data=iris)
>>> sns.swarmplot(x="species", y="petal_length", data=iris)
```

Scatterplot with one categorical variable

Categorical scatterplot with non-overlapping points

```
>>> sns.barplot(x="sex", y="survived", hue="class", data=titanic)
```

Show point estimates and confidence intervals with scatterplot glyphs

```
>>> sns.countplot(x="deck", data=titanic, palette="Greens_d")
```

Show count of observations

```
>>> sns.pointplot(x="class", y="survived", hue="sex", data=titanic, palette={"male": "g", "female": "m"}, markers=["^", "o"], linestyle=["-", "--"])
```

Show point estimates and confidence intervals as rectangular bars

```
>>> sns.boxplot(x="alive", y="age", hue="adult_male", data=titanic)
>>> sns.boxplot(data=iris, orient="h")
```

Boxplot

Boxplot with wide-form data

```
>>> sns.violinplot(x="age", y="sex", hue="survived", data=titanic)
```

Violin plot

Context Functions

```
>>> sns.set_context("talk")
>>> sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth":2.5})
```

Set context to "talk" Set context to "notebook", scale font elements and override param mapping

Color Palette

```
>>> sns.set_palette("husl", 3)
>>> sns.color_palette("husl")
>>> flatui = ["#9b59b6", "#3498db", "#95a5a6", "#e74c3c", "#34495e", "#2ecc71"]
>>> sns.set_palette(flatui)
```

Define the color palette Use with with to temporarily set palette Set your own color palette

```
>>> h = sns.PairGrid(iris)
>>> h = h.map(plt.scatter)
>>> sns.pairplot(iris)
>>> i = sns.JointGrid(x="x", y="y", data=data)
>>> i = i.plot(sns.regplot, sns.distplot)
>>> sns.jointplot("sepal_length", "sepal_width", data=iris, kind='kde')
```

Subplot grid for plotting pairwise relationships Plot pairwise bivariate distributions Grid for bivariate plot with marginal univariate plots

Plot bivariate distribution

Regression Plots

```
>>> sns.regplot(x="sepal_width", y="sepal_length", data=iris, ax=ax)
```

Plot data and a linear regression model fit

Distribution Plots

```
>>> plot = sns.distplot(data.y, kde=False, color="b")
```

Plot univariate distribution

Matrix Plots

```
>>> sns.heatmap(uniform_data, vmin=0, vmax=1)
```

Heatmap

4 Further Customizations

Also see Matplotlib

Axisgrid Objects

```
>>> g.despine(left=True)
>>> g.set_ylabels("Survived")
>>> g.set_xticklabels(rotation=45)
>>> g.set_axis_labels("Survived", "Sex")
>>> h.set(xlim=(0,5), ylim=(0,5), xticks=[0,2.5,5], yticks=[0,2.5,5])
```

Remove left spine Set the labels of the y-axis Set the tick labels for x Set the axis labels

Set the limit and ticks of the x-and y-axis

Plot

```
>>> plt.title("A Title")
>>> plt.ylabel("Survived")
>>> plt.xlabel("Sex")
>>> plt.ylim(0,100)
>>> plt.xlim(0,10)
>>> plt.setp(ax, yticks=[0,5])
>>> plt.tight_layout()
```

Add plot title Adjust the label of the y-axis Adjust the label of the x-axis Adjust the limits of the y-axis Adjust the limits of the x-axis Adjust a plot property Adjust subplot params

5 Show or Save Plot

Also see Matplotlib

```
>>> plt.show()
>>> plt.savefig("foo.png")
>>> plt.savefig("foo.png", transparent=True)
```

Show the plot Save the plot as a figure Save transparent figure

Close & Clear

Also see Matplotlib

```
>>> plt.cla()
>>> plt.clf()
>>> plt.close()
```

Clear an axis Clear an entire figure Close a window



## Bokeh

Learn Bokeh **Interactively** at [www.DataCamp.com](https://www.datacamp.com),  
taught by Bryan Van de Ven, core contributor

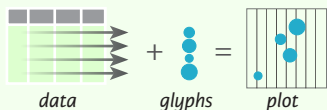


### Plotting With Bokeh

The Python interactive visualization library **Bokeh** enables high-performance visual presentation of large datasets in modern web browsers.



Bokeh's mid-level general purpose `bokeh.plotting` interface is centered around two main components: data and glyphs.



The basic steps to creating plots with the `bokeh.plotting` interface are:

1. Prepare some data:
2. Create a new plot
3. Add renderers for your data, with visual customizations
4. Specify where to generate the output
5. Show or save the results

```
>>> from bokeh.plotting import figure
>>> from bokeh.io import output_file, show
>>> x = [1, 2, 3, 4, 5]
>>> y = [6, 7, 2, 4, 5]
>>> p = figure(title="simple line example",
>>>             x_axis_label='x',
>>>             y_axis_label='y')
>>> p.line(x, y, legend="Temp.", line_width=2)
>>> output_file("lines.html")
>>> show(p)
```

## 1 Data

Also see [Lists, NumPy & Pandas](#)

Under the hood, your data is converted to Column Data Sources. You can also do this manually:

```
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(np.array([[33.9, 4, 65, 'US'],
>>>                             [32.4, 4, 66, 'Asia'],
>>>                             [21.4, 4, 109, 'Europe']]
>>>                  columns=['mpg', 'cyl', 'hp', 'origin'],
>>>                  index=['Toyota', 'Fiat', 'Volvo'])
>>> from bokeh.models import ColumnDataSource
>>> cds_df = ColumnDataSource(df)
```

## 2 Plotting

```
>>> from bokeh.plotting import figure
>>> p1 = figure(plot_width=300, tools='pan,box_zoom')
>>> p2 = figure(plot_width=300, plot_height=300,
>>>             x_range=(0, 8), y_range=(0, 8))
>>> p3 = figure()
```

## 3 Renderers & Visual Customizations

### Glyphs

```
Scatter Markers
>>> p1.circle(np.array([1,2,3]), np.array([3,2,1]),
>>>           fill_color='white')
>>> p2.square(np.array([1.5,3.5,5.5]), [1,4,3],
>>>           color='blue', size=1)

Line Glyphs
>>> p1.line([1,2,3,4], [3,4,5,6], line_width=2)
>>> p2.multi_line(pd.DataFrame([[1,2,3],[5,6,7]]),
>>>               pd.DataFrame([[3,4,5],[3,2,1]]),
>>>               color="blue")
```

### Customized Glyphs

Also see [Data](#)

```
Selection and Non-Selection Glyphs
>>> p = figure(tools='box_select')
>>> p.circle('mpg', 'cyl', source=cds_df,
>>>          selection_color='red',
>>>          nonselection_alpha=0.1)

Hover Glyphs
>>> from bokeh.models import HoverTool
>>> hover = HoverTool(tooltips=None, mode='vline')
>>> p3.add_tools(hover)

Colormapping
>>> from bokeh.models import CategoricalColorMapper
>>> color_mapper = CategoricalColorMapper(
>>>               factors=['US', 'Asia', 'Europe'],
>>>               palette=['blue', 'red', 'green'])
>>> p3.circle('mpg', 'cyl', source=cds_df,
>>>           color=dict(field='origin',
>>>                       transform=color_mapper),
>>>           legend='Origin')
```

### Legend Location

```
Inside Plot Area
>>> p.legend.location = 'bottom_left'

Outside Plot Area
>>> from bokeh.models import Legend
>>> r1 = p2.asterisk(np.array([1,2,3]), np.array([3,2,1]))
>>> r2 = p2.line([1,2,3,4], [3,4,5,6])
>>> legend = Legend(items=[("One", [p1, r1]), ("Two", [r2])],
>>>                  location=(0, -30))
>>> p.add_layout(legend, 'right')
```

### Legend Orientation

```
>>> p.legend.orientation = "horizontal"
>>> p.legend.orientation = "vertical"
```

### Legend Background & Border

```
>>> p.legend.border_line_color = "navy"
>>> p.legend.background_fill_color = "white"
```

### Rows & Columns Layout

```
Rows
>>> from bokeh.layouts import row
>>> layout = row(p1,p2,p3)

Columns
>>> from bokeh.layouts import columns
>>> layout = column(p1,p2,p3)

Nesting Rows & Columns
>>> layout = row(column(p1,p2), p3)
```

### Grid Layout

```
>>> from bokeh.layouts import gridplot
>>> row1 = [p1,p2]
>>> row2 = [p3]
>>> layout = gridplot([[p1,p2],[p3]])
```

### Tabbed Layout

```
>>> from bokeh.models.widgets import Panel, Tabs
>>> tab1 = Panel(child=p1, title="tab1")
>>> tab2 = Panel(child=p2, title="tab2")
>>> layout = Tabs(tabs=[tab1, tab2])
```

### Linked Plots

#### Linked Axes

```
>>> p2.x_range = p1.x_range
>>> p2.y_range = p1.y_range
```

#### Linked Brushing

```
>>> p4 = figure(plot_width = 100,
>>>             tools='box_select,lasso_select')
>>> p4.circle('mpg', 'cyl', source=cds_df)
>>> p5 = figure(plot_width = 200,
>>>             tools='box_select,lasso_select')
>>> p5.circle('mpg', 'hp', source=cds_df)
>>> layout = row(p4,p5)
```

## 4 Output & Export

### Notebook

```
>>> from bokeh.io import output_notebook, show
>>> output_notebook()
```

### HTML

#### Standalone HTML

```
>>> from bokeh.embed import file_html
>>> from bokeh.resources import CDN
>>> html = file_html(p, CDN, "my_plot")
```

```
>>> from bokeh.io import output_file, show
>>> output_file('my_bar_chart.html', mode='cdn')
```

#### Components

```
>>> from bokeh.embed import components
>>> script, div = components(p)
```

### PNG

```
>>> from bokeh.io import export_png
>>> export_png(p, filename="plot.png")
```

### SVG

```
>>> from bokeh.io import export_svgs
>>> p.output_backend = "svg"
>>> export_svgs(p, filename="plot.svg")
```

## 5 Show or Save Your Plots

```
>>> show(p1)
>>> save(p1)
```

```
>>> show(layout)
>>> save(layout)
```



# Python For Data Science Cheat Sheet

## Jupyter Notebook

Learn More Python for Data Science Interactively at [www.DataCamp.com](https://www.datacamp.com)



### Saving/Loading Notebooks

Create new notebook

Make a copy of the current notebook

Save current notebook and record checkpoint

Preview of the printed notebook

Close notebook & stop running any scripts

Open an existing notebook

Rename notebook

Revert notebook to a previous checkpoint

Download notebook as

- IPython notebook
- Python
- HTML
- Markdown
- reST
- LaTeX
- PDF

### Writing Code And Text

Code and text are encapsulated by 3 basic cell types: markdown cells, code cells, and raw NBConvert cells.

#### Edit Cells

Cut currently selected cells to clipboard

Paste cells from clipboard above current cell

Paste cells from clipboard on top of current cell

Revert "Delete Cells" invocation

Merge current cell with the one above

Move current cell up

Adjust metadata underlying the current notebook

Remove cell attachments

Paste attachments of current cell

Copy cells from clipboard to current cursor position

Paste cells from clipboard below current cell

Delete current cells

Split up a cell from current cursor position

Merge current cell with the one below

Move current cell down

Find and replace in selected cells

Copy attachments of current cell

Insert image in selected cells

#### Insert Cells

Add new cell above the current one

Add new cell below the current one

### Working with Different Programming Languages

Kernels provide computation and communication with front-end interfaces like the notebooks. There are three main kernels:

IP[y]:  
IPython

R  
IRkernel

IJ[.]:  
IJulia

Installing Jupyter Notebook will automatically install the IPython kernel.

Restart kernel

Restart kernel & run all cells

Restart kernel & run all cells

Interrupt kernel

Interrupt kernel & clear all output

Connect back to a remote notebook

Run other installed kernels

### Command Mode:

### Edit Mode:

### Executing Cells

Run selected cell(s)

Run current cells down and create a new one above

Run all cells above the current cell

Change the cell type of current cell

toggle, toggle scrolling and clear all output

Run current cells down and create a new one below

Run all cells

Run all cells below the current cell

toggle, toggle scrolling and clear current outputs

### View Cells

Toggle display of Jupyter logo and filename

Toggle line numbers in cells

Toggle display of toolbar

Toggle display of cell action icons:

- None
- Edit metadata
- Raw cell format
- Slideshow
- Attachments
- Tags

### Widgets

Notebook widgets provide the ability to visualize and control changes in your data, often as a control like a slider, textbox, etc.

You can use them to build interactive GUIs for your notebooks or to synchronize stateful and stateless information between Python and JavaScript.

Download serialized state of all widget models in use

Save notebook with interactive widgets

Embed current widgets

1. Save and checkpoint
2. Insert cell below
3. Cut cell
4. Copy cell(s)
5. Paste cell(s) below
6. Move cell up
7. Move cell down
8. Run current cell
9. Interrupt kernel
10. Restart kernel
11. Display characteristics
12. Open command palette
13. Current kernel
14. Kernel status
15. Log out from notebook server

### Asking For Help

Walk through a UI tour

Edit the built-in keyboard shortcuts

Description of markdown available in notebook

Python help topics

NumPy help topics

Matplotlib help topics

Pandas help topics

List of built-in keyboard shortcuts

Notebook help topics

Information on unofficial Jupyter Notebook extensions

IPython help topics

SciPy help topics

SymPy help topics

About Jupyter Notebook

