PYTHON

FOR DATA SCIENCE

CHEAT SHEET

VARIABLES IN PYTHON

In Python, you'll work with variables a lot.
You can assign a value to a variable as simply as:

```
variable_name = variable_value
```

If you assign a new value to a variable that you have used before, it will overwrite your previous value.

Examples:

```
a = 100
b = 'some_random_text'
c = True
d = 0.75
```

[your notes]

BASIC DATA TYPES

In Python, we have quite a few different data types. But these four are the most important ones (for now):

1. **Integer.** A whole number without a fractional part. E.g. 100, 156, 2012412
2. **Float.** A number with a fractional part. E.g. 0.75, 3.1415, 961.1241250, 7/8
3. **Boolean.** A binary value. It can be either True or False.
4. **String.** It's a sequence of Unicode characters (e.g. numbers, letters, punctuation). It can be alphabetical letters only — or a mix of letters, numbers and other characters. In Python, it's easy to identify a string since it has to be between apostrophes (or quotation marks). E.g. 'hello', 'R2D2', 'Tomi', '124.56.128.41'

ARITHMETIC OPERATORS

Let's assign two values!
a = 3
b = 4

The arithmetic operations you can do with them:

Artihmetic operator	What does it do?	Result in our example
a + b	adds **a** to **b**	7
a - b	subtracts **b** from **a**	-1
a * b	multiplies **a** by **b**	12
a / b	divides **a** by **b**	0.75
b % a	divides **b** by **a** and returns remainder	1
a ** b	raises **a** to the the power of **b**	81

DATA STRUCTURES

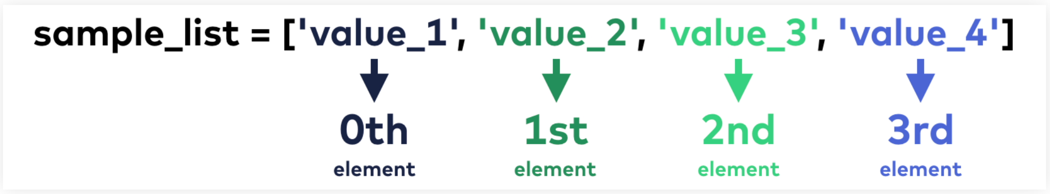
Data structures exist to organize your data and store related/similar data points in one "place." There are four data structure types. The two most important in data science are: **lists** and **dictionaries**.

#1: LISTS

A list is a sequence of values. It can store integers, strings, booleans, anything - even a mix of these.
Example:
sample_list = ['value1', 'value2', 'value3', 'value4', 1, 2, 3, 4, True, False]

Querying an element of a list:
sample_list[3]

IMPORTANT! Python works with zero-based indexing. E.g.


sample_list = ['value_1', 'value_2', 'value_3', 'value_4']

Example:
sample_list[3] — (This returns 'value4'.)

#2: DICTIONARY

A dictionary is a collection of key-value pairs. (Key is a unique identifier, value is the actual data.)

Example:

```
sample_dict = {'apple': 3,
               'tomato': 4,
               'orange': 1,
               'banana': 14,
               'is_store_open': True}
```

Querying an element of a dictionary:

sample_dict['banana'] — (This returns **14**.)

NESTED LISTS AND/OR DICTIONARY

You can create nested lists and dictionaries.

Example 1 (list within a list):

```
nested_list = ['val1', 'val2', ['nested_val1', 'nested_val2', 'nested_val3']]
```

Querying an element from the nested part:

nested_list[2][0] — (This returns **'nested_val1'**.)

Example 2 (list within a dictionary):

```
nested_dict = {'key_a': ['nested_val1', 'nested_val2', 'nested_val3'],
               'key_b': 'val2',
               'key_c': 'val3'}
```

Querying an element from the nested part:

nested_dict['key_a'][0] — (This returns **'nested_val1'**.)

FUNCTIONS AND METHODS

You can run functions and methods on your Python objects. Most functions and methods are designed to perform a single action on your input and transform it into a (different) output.

Example:
my_input = 'Hello'
len(my_input)
Output: 5 (That's the number of characters in 'Hello'.)

Calling a Python function looks like this: **function_name(arguments)**
Calling a Python method looks like this: **input_value.method_name(arguments)**

More details on the difference between functions and methods:
https://data36.com/python-functions

THE MOST IMPORTANT BUILT-IN FUNCTIONS

Let's assign a variable: **my_variable = 'Hello, World!'**

print(my_variable)
This prints the value of my_variable to the screen.
Output: **Hello, World!**

len(my_variable)
This returns the number of characters in a string - or the number of elements in a list. Output: **13** (That's the number of characters in 'Hello, World!')

type(my_variable)
This returns the data type of my_variable.
Output: **str** (That stands for string which is the data type of 'Hello, World!')

Find more Python functions here: https://data36.com/python-functions

THE MOST IMPORTANT METHODS FOR PYTHON STRINGS

Let's assign a variable: **my_variable = 'Hello, World!'**

my_variable.upper()

This returns the uppercase version of a string.

Output: **'HELLO, WORLD!'**

my_variable.lower()

This returns the lowercase version of a string.

Output: **'hello, world!'**

my_variable.split(',')

This splits your string into a list. The argument specifies the separator that you want to use for the split.

Output: **['Hello', ' World']**

my_variable.replace('World', 'Friend')

This replaces a given string with another string. Note that it's case sensitive.

Output: **'Hello, Friend!'**

THE MOST IMPORTANT METHODS FOR PYTHON LISTS

Let's make a list: **my_list = [10, 131, 351, 197, 10, 148, 705, 18]**

my_list.append('new_element')

It adds an element to the end of your list. The argument is the new element itself. This method updates your list and it doesn't have any output.

> If you query the list after running this method:
> **my_list**
> Output: **[10, 131, 351, 197, 10, 148, 705, 18, 'new_element']**

my_list.remove(10)

It removes the first occurrence of the specified element from your list. This method updates your list and it doesn't have any output.

my_list
Output: **[131, 351, 197, 10, 148, 705, 18, 'new_element']**

my_list.clear()

It removes all elements of the list. This method updates your list and it doesn't have any output.

my_list
Output: **[]**

Find more Python functions and methods here:
https://data36.com/python-functions

All Python built-in functions:
https://docs.python.org/3/library/functions.html

All Python string methods:
https://docs.python.org/3/library/stdtypes.html#string-methods

All Python list methods:
https://docs.python.org/3/tutorial/datastructures.html

IMPORTANT! These are only the built-in Python functions and methods. You can get access to many more with the import statement. (See page 12!)

IF STATEMENT

If statements are great for evaluating a condition and taking certain action(s) based on the result.

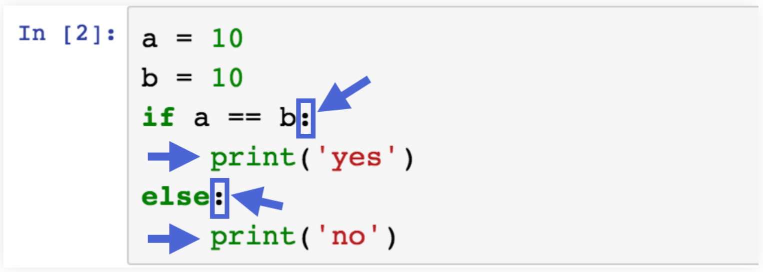
Example:

```
a = 10
b = 20
if a == b:
  print('yes')
else:
  print('no')
```

[your notes]

IMPORTANT! Be really careful with the syntax.

1. Never skip the colons at the end of the if and else lines!
2. Never skip the indentation (a tab or four spaces) at the beginning of the statement-lines!



```
In [2]: a = 10
        b = 10
        if a == b:
            print('yes')
        else:
            print('no')
```

In the if line (condition) you can use comparison and logical operators.
Let's see them. Assign four values!

a = 3
b = 4
c = True
d = False

Comparison operator	What does it evaluate?	Result in our example
a == b	if **a** equals **b**	False
a != b	if **a** doesn't equal **b**	True
a < b	if **a** is less than **b**	True
a > b	if **a** is greater than **b**	False

Logical operator	What does it evaluate?	Result in our example
c and d	if both **c** and **d** are **True**	False
c or d	if either **c** or **d** is **True**	True
not c	returns the opposite of **c**	False

IF STATEMENT WITH MORE COMPLEX CONDITIONS

The condition can be complex.

Example:

```
a = 10
b = 20
c = 30
if (a + b) / c == 1 and c - b - a == 0:
   print('yes')
else:
   print('no')
```

IF-ELIF-ELSE STATEMENT

You can use condition-sequences to evaluate multiple conditions.

Example:

```
a = 10
b = 11
c = 10
if a == b:
   print('a equals b, nice')
elif a == c:
   print('a equals c, nice')
else:
   print('a equals nothing... too bad')
```

Note: You can add as many elifs as you need.

FOR LOOPS

For loops are for iterating through iterables (e.g. lists, strings, range() objects) and taking certain action(s) on the individual elements of the given iterable.

Example:

```
sample_list = ['value1', 'value2', 'value3', 'value4', 1, 2, 3, 4, True, False]
for i in sample_list:
    print(i)
```

Output:

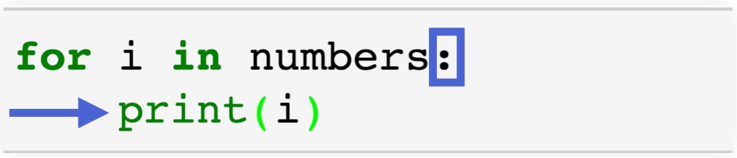
```
value1
value2
value3
value4
1
2
3
4
True
False
```

[your notes]

The action itself can be anything, not just **print()**. (Even multiple actions.)

IMPORTANT! Be really careful with the syntax.

1. Never skip the colons at the end of the for line!
2. Never skip the indentations (tabs or four spaces) in the body of the loop!



```
for i in numbers:
    print(i)
```

FOR LOOPS (WITH range() OBJECTS)

If you want to iterate through numbers, you can use **range()**.

Example 1:

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

range() is a function. It accepts three (optional) arguments: start, stop, step.

Example 2:

```
for i in range(100,200,20):
    print(i)
```

Output:

```
100
120
140
160
180
```

More about for loops: https://data36.com/python-for-loops

NESTED FOR LOOPS + FOR LOOPS AND IF STATEMENTS COMBINED

You can combine for loops with for loops (called nested for loops).
And you can combine for loops and if statements.

I wrote more about these here:
https://data36.com/python-nested

PYTHON FORMATTING TIPS & BEST PRACTICES

1) ADD COMMENTS WITH THE # CHARACTER!

Example:

```
# This is a comment before my for loop.
for i in range(0, 100, 2):
   print(i)
```

2) VARIABLE NAMES

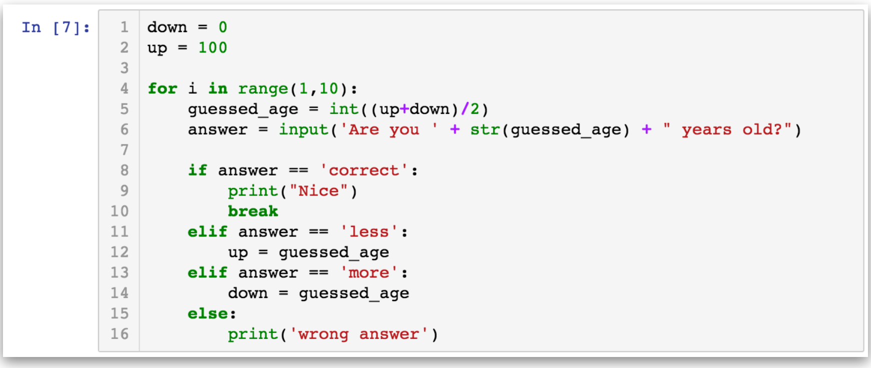
Conventionally, variable names should be written with lowercase letters, and the words in them separated by _ characters. Make sure that you choose meaningful and easy-to-distinguish variable names!

Example:

```
my_meaningful_variable_name = 100
```

3) USE BLANK LINES TO SEPARATE CODE BLOCKS VISUALLY!

Example:



```
In [7]:
down = 0
up = 100

for i in range(1,10):
    guessed_age = int((up+down)/2)
    answer = input('Are you ' + str(guessed_age) + " years old?")

    if answer == 'correct':
        print("Nice")
        break
    elif answer == 'less':
        up = guessed_age
    elif answer == 'more':
        down = guessed_age
    else:
        print('wrong answer')
```

4) USE WHITE SPACES AROUND OPERATORS AND ASSIGNMENTS!

Good example:

```
number_x = 10
number_y = 100
number_mult = number_x * number_y
```

Bad example:

```
number_x=10
number_y=100
number_mult=number_x*number_y
```

IMPORTING OTHER PYTHON MODULES AND PACKAGES

Use the **import** statement to expand the original Python3 toolset with additional modules and packages.

General syntax:
import [module_name]

Or:
from [module_name] import [item_name]

THE MOST IMPORTANT BUILT-IN MODULES FOR DATA SCIENTISTS

RANDOM

Examples:
import random
This imports the random module. (No output.)

random.random()
This generates a random float between 0 and 1. (Output example: **0.6197724959**)

random.randint(1,10)
This generates a random integer between 1 and 10. (Output example: **4**)

STATISTICS

Examples:

import statistics

This imports the statistics module.

```
my_list = [0, 1, 1, 3, 4, 9, 15]
statistics.mean(my_list)
statistics.median(my_list)
statistics.mode(my_list)
statistics.stdev(my_list)
statistics.variance(my_list)
```

These calculate the mean, median, mode, standard deviation and variance for the list called my_list. (Note: You have to run them one by one.)

MATH

Examples:

import math

This imports the math module.

math.factorial(5)

This returns 5 factorial. (Output: **120**)

math.pi

This returns the value of pi. (Output: **3.141592653589793**)

math.sqrt(5)

This returns the square root of 5. (Output: **2.23606797749979**)

DATETIME

Python3, by default, does not handle dates and times. But if you import the datetime module, you will get access to these functions, too.

Example:

import datetime

This imports the datetime module.

datetime.datetime.now()

This returns the current date and time in tuple format. (Note: A tuple is like a list, but can't be changed.)

Output: **datetime.datetime(2019, 7, 14, 0, 46, 30, 906311)**

datetime.datetime.now().strftime("%F")

This returns the current date and time in the usual yyyy-mm-dd format.

Output: **'2019-07-14'**

CSV

This module helps you to open and manage .csv files in Python.

Example:

```
import csv
with open('example.csv') as csvfile:
  my_csv_file = csv.reader(csvfile, delimiter=';')
  for row in my_csv_file:
    print(row)
```

These few lines import the csv module and then open the example.csv file - where the fields are separated with semicolons (;). The last two lines of the code print all the rows (of the that .csv file we opened) one by one.

MORE INFO ABOUT THE PYTHON BUILT-IN MODULES

- https://data36.com/python-import/
- https://docs.python.org/3/library/random.html
- https://docs.python.org/3/library/statistics.html
- https://docs.python.org/3/library/math.html
- https://docs.python.org/3/library/datetime.html
- https://docs.python.org/3/library/csv.html

THE 5 MOST IMPORTANT "EXTERNAL" PYTHON LIBRARIES AND PACKAGES FOR DATA SCIENTISTS

- **Numpy**
- **Pandas**
- **Matplotlib**
- **Scikit-Learn**
- **Scipy**

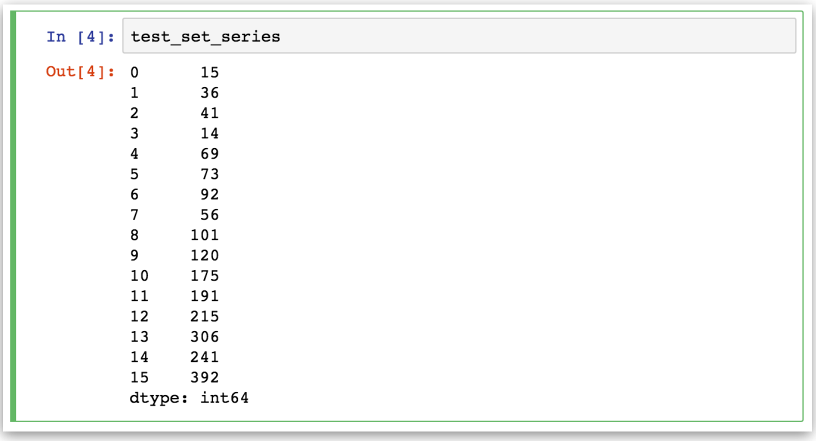
PANDAS

Pandas is one of the most popular Python libraries for data science and analytics. It helps you manage two-dimensional data tables and other data structures. It relies on Numpy, so when you import Pandas, you need to import Numpy first.

import numpy as np
import pandas as pd

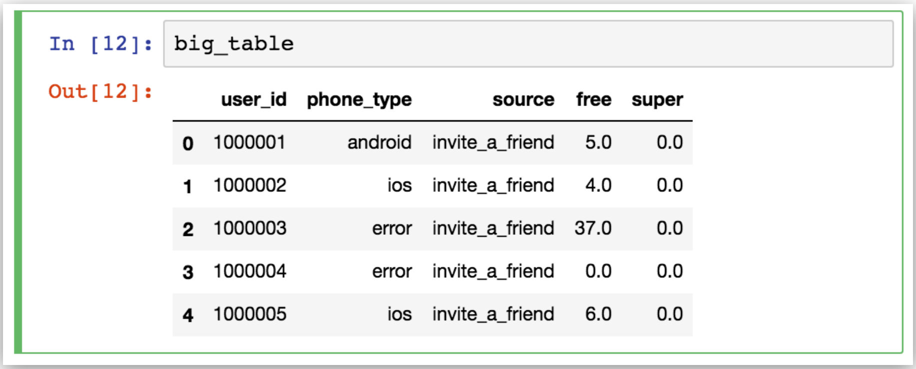
PANDAS DATA STRUCTURES

Series: Pandas Series is a one dimensional data structure ("a one dimensional ndarray") that can store values, with a unique index for each value.



```
In [4]: test_set_series
Out[4]: 0       15
        1       36
        2       41
        3       14
        4       69
        5       73
        6       92
        7       56
        8      101
        9      120
        10     175
        11     191
        12     215
        13     306
        14     241
        15     392
        dtype: int64
```

DataFrame: Pandas DataFrame is a two (or more) dimensional data structure – basically a table with rows and columns. The columns have names and the rows have indexes.



```
In [12]: big_table
Out[12]:
```

	user_id	phone_type	source	free	super
0	1000001	android	invite_a_friend	5.0	0.0
1	1000002	ios	invite_a_friend	4.0	0.0
2	1000003	error	invite_a_friend	37.0	0.0
3	1000004	error	invite_a_friend	0.0	0.0
4	1000005	ios	invite_a_friend	6.0	0.0

OPENING A .CSV FILE IN PANDAS

pd.read_csv('/home/your/folder/file.csv', delimiter=';')
This opens the .csv file that's located in /home/your/folder and called file.csv. The fields in the file are separated with semicolons (;).

df = pd.read_csv('/home/your/folder/file.csv', delimiter=';')
This opens a .csv file and stores the output into a variable called df. (The variable name can be anything else - not just df.)

pd.read_csv('file.csv', delimiter=';', names = ['column1', 'column2', 'column3'])
This opens file.csv. The fields in the file are separated with semicolons (;). We change the original names of the columns and set them to: 'column1', 'column2' and 'column3'.

QUERYING DATA FROM PANDAS DATAFRAMES

df
It returns the whole dataframe. (Note: remember, when we opened the .csv file, we stored our dataframe into the **df** variable!)

df.head()
It returns the first 5 rows of **df**.

df.tail()
It returns the last 5 rows of **df**.

df.sample(7)
It returns 7 random rows from **df**.

df[['column1', 'column2']]
It returns column1 and column2 from **df**. (The output is in DataFrame format.)

df.column1
It returns column1 from **df**. (The output is in Series format.)

df[my_dataframe.column1 == 'given_value']
It returns all columns, but only those rows in which the value in column1 is 'given_value'. (The output is in DataFrame format.)

df[['column1']][my_dataframe.column1 == 'given_value'].head()
It takes the column1 column — and only those rows in which the value in column1 is 'given_value' — and returns only the first 5 rows. (The point is: you can combine things!)

AGGREGATING IN PANDAS

The most important pandas aggregate functions:

- **.count()**
- **.sum()**
- **.mean()**
- **.median()**
- **.max()**
- **.min()**

Examples:
df.count()
It counts the number of rows in each column of **df**.

df.max()
It returns the maximum value from each column of **df**.

df.column1.max()
It returns the maximum value only from the column1 column of **df**.

PANDAS GROUP BY

The **.groupby()** operation is usually used with an aggregate function (**.count()**, **.sum()**, **.mean()**, **.median()**, etc.). It groups the rows by a given column's values. (The column is specified as the argument of the **.groupby()** operation.) Then we can calculate the aggregate for each group and get that returned to the screen.

df.groupby('column1').count()
It counts the number of values in each column - for each group of unique column1 values.

df.groupby('column1').sum()
It sums the values in each column - for each group of unique column1 values.

df.groupby('column1').min()
It finds the minimum value in each column - for each group of unique column1 values.

df.groupby('column1').max()
It finds the maximum value in each column - for each group of unique column1 values.

A FEW MORE USEFUL PANDAS METHODS

df.merge(other_df)
It joins **df** and **other_df** - for every row where the value of column1 from **df** equals the value of column1 from **other_df**.

df.merge(other_df, how = 'inner', left_on = 'col2', right_on = 'col6')
It joins **df** and **other_df** - for every row where the value of 'col2' from **df** ("left" table) equals the value of 'col6' from **other_df** ("right" table). The join type is an inner join.

df.sort_values('column1')
It returns every row and column from **df**, sorted by column1, in ascending order (by default).

df.sort_values('column1', ascending = False)
It returns every row and column from **df**, sorted by column1, in descending order.

df.sort_values('column1', ascending = False).reset_index(drop = True)
It returns every row and column from **df**, sorted by column1, in descending order. After sorting, it re-indexes the table: removes the old indexes and sets new ones.

df.fillna('some_value')
It finds all empty (NaN) values in **df** and replaces them with 'some_value'.

Great pandas cheatsheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf