Python Reference

by : Dwight

basic syntax

pandas

matplotlib

spark

Python Basics

Legend
- notes
- code
- explanation

variable assignment

In python, there is no need to assign a type to a variable. For instance,
code:

```
spam amount = 5
```
(spam amount: variable; 5: value)

Comments

There are two different types of comments:

i) in-line:

```
# this is an inline comment
```

ii) code block comment / docstring

```
""" this is a code block comment """
```

Type

There are 4 basic python types:
int (whole number), float (decimals), string (words), boolean (true / false)

to check the type of a variable,
code:

```
type( word_count )
```

if-statements:

Note, python uses blank space to indicate what statements belong to what block of code.
↳ 4 spaces indicates a block of code

syntax:

```
if <expression> :
    statement
elif <expression> :
    statement
else :
    statement
```
(4 spaces)

numbers & numerical operations

There are two different types of numbers in python: integers and floats. Here are some operations you can do on numbers:

Name	Operator	Description
addition	a+b	sum a & b
subtraction	a-b	difference of a & b
multiply	a * b	product of a & b
division	a/b	quotient of a & b
floor division	a // b	quotient of a & b, removing fractions

modulus	a % b	integer remainder after division
exponent	a**b	a raised to the power of b

order of operations:
all math expressions are evaluated as follows,

Parentheses
Exponent
Multiplication
Division
Addition
Subtraction

Note: floats have interesting methods associated with them. For instance:

```
x = 0.125
x.as_integer_ratio()      ... output: (1,8)
numerator, denominator = x.as_integer_ratio()
```

double variable assignment: numerator = 1 and denominator = 8

aggregations:

- absolute value
  `abs(-32)` ... output: 32
- minimum
  `min(1,2,3)` ... output: 1
- maximum
  `max(1,2,3)` ... output: 3

casting values:
we can force python to change the type of a variable or python

for example,

```
print( float(10) )
output: 10.0

print( int 3) )
output: 3
```

functions

the most basic fcn we can use is the helper function
↳ helps you understand how a python fcn works

code:

```
help(round)
```

name of the fcn and not the function call. For instance, round() is a fcn call

defining a fcn syntax:

```
def <fcn_name> (arg):
    <code block>
    return expression
```

we can also put default values in our fcn so that if user doesn't enter anything it does a calculation based on default

It is best practice to put a docstring in a fcn

fcns with a docstring:

```
def <fcn_name> (arg):
    """ what the fcn does

    >>> fcn_name()
    output
    """
    <code block>
    return <expression>
```

This is a docstring what we see when we call the help fcn

boolean

Booleans are logical statement evaluations; either true or false

Operators:

Name	logical equal	less than	less than equal to	not equal to	greater than	greater than equal to
Operator	a == b	a < b	a <= b	a != b	a > b	a >= b

other operators:
and, or, not

truth tables:

And

Arg1	Arg2	Result
True	True	True
True	False	False
False	True	False
False	False	False

Or

Arg1	Arg2	Result
True	True	True
True	False	True
False	True	True
False	False	False

boolean conversion:
bool(1) . . . all numbers are true, except zero
bool("asf") . . . all string true except empty string

lists

an ordered sequence of values

code
```
prime = [2, 3, 5, 7]
```

indexing:
Note, in python the list starts at index zero

code
```
prime[0]    . . . output: 2    ; first element of list
prime[-1]   . . . output: 7    ; last element of list
```

slicing:
using select parts of a list
code
```
prime[0:3]    . . . first element until the 3rd element
prime[ :3]    . . . leave out the end index but, give me every element until that point
prime[3: ]    . . . all elements from the 4th element
prime[-3:]    . . . last 3 elements of list
```

changing elements in a list:
```
prime[0] = "Alex"
prime
```

Output:
[Alex, 3, 5, 7]

list functions:
len(prime) . . . length of a list
sum(prime) . . . sum the list

list operators

operator	example
in	5 in prime

list methods:
a method is a fun attached to an object

```
prime.append(9)          . . . add element to end of list
prime.pop(9)             . . . removes last element of list
prime.index("Earth")     . . . find the index of an element
prime.count(5)           . . . count number of occurences of 5
<list>.extend(<list>)    . . . add a list to the end of a list
```

Tuples

idea: an immutable/unchangable list

example:

```
t = (1, 2, 3)
t = 1, 2, 3
```

] different ways of doing the same thing

List Comprehensions

A way of doing operations on a list as you are creating it

```
square = [n**2 for n in range(10)]
```

This executes last saying to square the list. This executes first saying: make a list from 0–9. . . output: [0, 1, 4, 9, 16, 25, ... 81]

```
short_planet = [planet for planet in planets if len(planet) < 6]
```

① a list of planets whose names are shorter than 6 characters

② what happens when the condition is met

③

Loops

for loop syntax:

```
for <element> in <list>:
    <code block>
```

range loop syntax: (a range up to but not including this number)

```
for i in range(5):
    <code block>
```

while loop:

```
while <boolean condition>:
    <code block>
```

sets:

idea: a list where order doesn't matter.

downfall: can't rely on the order of info when it is retrieved

```
my_set = {1, 2, 3}
```

→ can only hold immutable data types

Dictionaries

A list of key-value pairs

- we can also do comprehensions similar to list comprehensions, called dictionary comprehensions

```
numbers = {"one": 1, "two": 2}
```

indexing a dictionary:

```
numbers['one']
```

. . . output: 1

adding to a dictionary: (a key not in the dictionary)

```
numbers["eleven"] = 11
```

changing values in a dictionary:

```
numbers['one'] = 'sports'
```

Strings

A way of passing words into python

Similar to lists we can slice strings (look at list section for examples).

String methods:

- `<string>.capitalize()` . . . capitalize the first letter of a string
- `<string>.upper()` . . . capitalize the whole string
- `<string>.lower()` . . . make the whole string lowercase
- `<string>.index()` . . . search for the first index of a substring
- `<string>.split()` . . . turn a string into a list of words
- `<string>.count(arg)` . . . how many times the string <arg> occurs in the larger string
- `<string>.isdigit()` . . . determine if string contains only digits
- `<string>.isalpha()` . . . determine if string contains only words

```
"{} is unhappy on {}".format(name, weekday)
```

① ② predifined variables: insert these values in the string where there are curly brackets

Importing libraries

libraries add more functionality to base python

```
import math
print(dir(math))
```

. . . math is a module (aka a collection of variables)

. . . dir() allows use to see all the variables in a module

we can access variables using dot syntax:

```
pi = math.pi
```

modules also have fcns in them:

```
math.log(32,2)
help(math.log)
```

. . . tells us how to use the log fcn in math module

alias import statements:

```
import math as mt
```

. . . instead of using math we can use mt

mt.pi . . . same as calling math.pi bc mt is an alias for math

import all:

```
from math import *          . . . make all variables from the math module available
from math import log, pi    . . . we can also import only specific variables
from numpy import numpy.random   . . . we can also import sub libraries held in variables
```

Pandas

installing pandas

```
pip install pandas
or
conda install pandas
```

importing pandas

```
import pandas as pd
```

get a summary of data

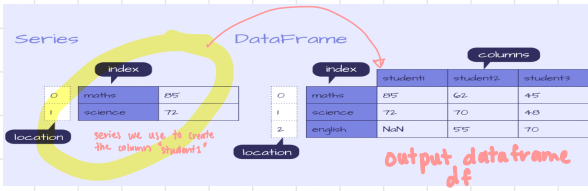
```
df.describe()
```

tells us about the mean, std deviation, min/max, IQR and # of non-missing values for every column of a dataframe

Pandas hosts 2 data structures:

i) series - a list / column of data

ii) dataframes - a table similar to that in spreadsheets



3 methods to create a dataframe:

i) empty dataframe:

```
df = pd.DataFrame()
```

ii) dataframe of series:

```
d = {'student1': pd.Series([85., 72.], index = ['maths', 'science']),
     'student2': pd.Series([62., 70., 55.], index = ['maths', 'science', 'english']),
     'student3': pd.Series([45., 48., 70.], index = ['maths', 'science', 'english'])}
```

(column values; row names [in-order from top-to-bottom]; 'student1' etc. = column name)

we can also use a dictionary of lists

```
df = pd.DataFrame(d)      . . . creates a dataframe from the dictionary of series above
print(df.head())          . . . show the first 5 entries of the table
```

iii) create a dataframe from a file

2 main ways to get a dataframe from a file:

↳ ① csv files:

```
df = pd.read_csv( csv_file_name )
```

(path to csv file or web address)

if we put the argument index_col = 0 after the filename we can index columns using numbers

② querying a database

```
import pymysql
con = pymysql.connect( host = 'localhost', user = 'test', password = '', db = 'penguinDB' )
df = read_sql( f """ select * from penguins """, con )
```

(database connection info; f"""...""" = docstring quotes, so we can use quotes in our query; select * from penguins = query; con = connect to database)

export dataframe to csv file:

```
df.to_csv('output.csv', compression = 'gzip')
```

('output.csv' = filename we want to create)

dimensions of a dataframe

```
len(df)              ... number of rows
len(df.columns)      ... number of columns
df.size              ... number of elements in the table (rows x columns)
df.shape             ... a tuple of format (rows, columns)
df.info()            ... info on column types (str, int, bool, etc.) and number of non-null entries
```

we can also set how many items we see when we call the head fcn

```
pd.set_option('display.max_columns', None)   ... permanently set # of items we see
```

option to set — how many columns to max out at

pandas types:

dtype	Description
object	str or mixed datatypes
int64	Default for integers
float64	Default for floats
bool	True/False used for missing values
datetime64	Datetime values
category	Limited, usually fixed number of possible values

accessing data

get the names of columns in the dataframe

```
df.columns
```

view a column

same thing:

i) `df.column_name`

ii) `df['column_name']`

column name as a string (can also be a list of columns we want to see. For example, `df['column1','column2']`)

get one value in a column

```
df['column_name'][index]
```

index: integer

removing individual/a group of columns

```
df.drop(columns = [column_names])
```

indexing in python (2 types):

↳ index-based selection (selection using #'s)

```
df.iloc[row_index]              ... return this row in df
df.iloc[:, column_index]        ... return this column
df.iloc[:3, column_index]       ... return a column up to this row, but not including it
df.iloc[[0,1,2], column_index]  ... return only the rows in the list
```

index range: first element of range is included. Last element of range excluded

row index

↳ location-based selection (select using column names)

```
df.loc[rows, cols]
```

example:

```
keep_cols = ["student1", "student3"]
df = df.loc[:, keep_cols]
df.loc['Apples':'Potatoes']
```

index range: first element of range included. Last element of range also included

index of rows

column names we want/index of columns

return all the columns alphabetically between 'Apples' & 'Potatoes'

conditional selection of rows

Only view rows & columns that satisfy some condition

```
df.country == 'Italy'
```
... returns a column of booleans where True means that row satisfies the condition

```
df.loc[df.country == 'Italy']
```
... return a dataframe where all rows satisfy this condition

```
df.loc[ <condition 1> & <condition 2> ]   ... dataframe satisfying 2 conditions
df.loc[ <condition 1> | <condition2> ]    ... dataframe satisfying at least one of two conditions
df.loc[ df.column_name.isin(<list>) ]     ... only see rows where the value of the given column is in the given list
```

we also we can also use the fcn is null to get a table where the values in a column are null. For example:
`df.loc[df.column_name.isnull()]`

missing values — in pandas NaN = null

determine # of missing values in each column:

```
df.isna().sum()
```

or

```
df.isnull().sum()
```

remove missing values

```
df.dropna(inplace = True)
```

replace missing values

```
df['sex'].fillna('unknown', inplace = True)
```

what to put where there are missing values

column name

```
df.column_name.fillna('unknown')
```

replacing incorrect values:

```
df.column_name.replace(old_value, new_value)
```
(old_value: can be of any type)

when to remove:

when to replace:

data integrity: what values are in each column?

```
for col in ['Region', 'Island', 'sex']:
    print(f'Column: {col} has {df[col].nunique()} unique values: {df[col].unique()}')
```
(nunique(): # of unique values in the column; unique(): unique values in the column)

alternatively, we can call the value_counts method to see unique values in a column, as well as, the number of times each value appears in that column

```
df.column_name.value_counts()
```

making a column categorical

```
df['Species'] = df['Species'].astype('category')
```
('Species': column name; 'category': pandas data type)

advantages of make a column categorical? We save significant space in memory

maps

idea: take one set of values and map them to another set of values
↳ basically a transformation

2 methods to do a map:

i) using the map fcn:

```
df.column_name.map(lambda x: x - 3)
```
(x: input; x - 3: transformation; map fcn; return a new transformed Series/column)

. . . for every row in this column apply this transformation

ii) using the apply fcn:

```
df.apply(function, axis = 'columns')
df.apply(function, axis = 'index')
```
. . . apply a fcn to each row

. . . apply a fcn to each column

(return a new transformed dataframe)

groupwise analysis

Allows us to group rows by common values in a column

```
df.groupby('column_name')
```
. . . returns a dataframe we can do aggregations on

```
df.groupby('points').points.count()
```
. . . count number of unique values in the points column

```
df.groupby('points').price.min()
```
('points': column from dataframe)

. . . return the lowest price for every point rated

```
df.groupby('winery').apply(lambda table: table.title.iloc[0])
```
(table.title: column name, in this case title is the column containing wine names)

. . . the first wine every reviewed by every winery

Note, we can also groupby multiple columns; just pass a list in the argument of the groupby fcn

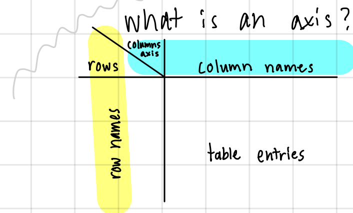
do multiple aggregations on a column at a time

```
df.groupby(['country']).price.agg([len, min, max])
```

fcns we want to apply to the price column

. . . returns a dataframe of aggregations given in the list

sort columns




```
df.sort(column= column_name, ascending = True)
```

sort by increasing value

renaming columns/row indices/axis names

```
df.rename(columns = {'old_column_name': 'new_column_name'})
```
. . . rename a column

```
df.rename(index = {old_index1 : new_index1})
```
. . . rename row indices

```
df.rename_axis('wines', axis = 'rows').rename_axis('fields', axis = 'columns')
```

change the name of row axis to this

change the name of the column axis to this

combining dataframes

Concatenation – when we have two tables with the same column names but different data

```
canadian_views = pd.read_csv(...)
british_views = pd.read_csv(...)
pd.concat([canadian_views, british_views])
```

] same column names in each table

join – dataframe must have the same number of rows

```
left = canadian_views.set_index(['title', 'trending_date'])
right = british_views.set_index(['title', 'trending_date'])

left.join(right, lsuffix = '_CAN', rsuffix = '_UK')
```

make these column indices for our data

table names; first columns we will see come from here

column names; second set of column names come from here

add this string to the end of all column names from the left table

add this string to all column names from the right table

Tidying a dataframe

melt = pivot longer
pivot = pivot wider
wide_to_long = seperate

https://pandas.pydata.org/docs/user_guide/reshaping.html

These are functions used to tidy our data. What does it mean to tidy a dataframe? Each observation has its own row, each variable has its own column and each type of observation is in one table.

pivot longer (aka melt):

the pivot_longer fcn from R is called melt in python

for example:

pigs dataframe

pig \ feed	feed1	feed2	feed3	feed4
1	60.8	68.7	92.6	87.9
2	57.0	67.7	92.1	84.2
3	65.0	74.0	90.2	83.1
4	58.6	66.3	96.2	85.7
5	61.7	64.8	99.1	90.3

table of pig weights by feed

```
pigs = pd.read_csv(...)
pigs['id'] = df.index
pigs.melt(id_vars =['id'], var_name = 'feed', value_name='weight')
```

. . . add a new column to the dataframe of row indices

name of column(s) we don't want to put in new column

a list

what we will call the new column that has the old column names

what to call the new column that has the values of the old column

alternatively,

```
pigs.melt(id_vars=['id'], value_vars = ['feed1', 'feed2', 'feed3', 'feed4'],
          var_name = 'feed', value_name = 'weight')
```

store in a variable if you want to keep this

list of columns we want to combine into one column

output:

id	feed	weight
0	feed 1	60.8
1	feed1	57.0
2	feed1	65.0
3	feed1	58.6
4	feed1	61.7
0	feed2	68.7
⋮	⋮	⋮

QED □

pivot longer & seperate

Alternatively, it could be the case that a wide table has many variables that represent two items. For example:

tb dataframe

country code: iso2	year	m04: males between ages 0 - 4	m514: males between ages 5 & 14	f04: females between ages 0 and 4
AD	1996	0	0	1
CA	1996	3	5	8
US	1994	1	10	2

untidy data: should have different column for age & gender

```
tb = read_csv( ... )
tb2 = pd.wide_to_long(tb, ['m', 'f'], i='iso2', j='age',
      sep = 1 )
```

- this is also where values from these columns of the old table will be stored
- column to use as id variable
- name of new column(s) being created that will contain the part of the column title we cut off
- index of the column name we want to seperate from; can also be a character like "_"
- the characters the columns we want to split up starts with

output:

iso2	year	m	f	age
AD	1996	0	1	04
AD	1996	0	null	514
CA	1996	3	8	04
CA	1996	5	null	514
US	1994	1	2	04
US	1994	10	null	514

Note, this is not yet tidy but we are close

tb_clean dataframe

iso2	year	gender	age	frequency
AD	1996	m	04	0
		f	04	1
		m	514	0
		f	514	null
CA		m	04	3
		f	04	8
		m	514	5
		f	514	null
US	1994	m	04	1
⋮	⋮	⋮	⋮	⋮

```
tb_clean = tb2.melt(id_vars=['iso2', 'year', 'age'],
           value_vars=['m', 'f'], var_name = 'gender',
           value_name = 'frequency')
```

- what remains constant (in terms of columns)
- what columns we want to combine
- name of new column that contains old column names
- new column name where values will be stored

QED □

pivot wider

Similar to seperate, used when one column holds multiple variables.

table "df"

	countries	metrics	values
0	A	population_in_million	100
1	B	population_in_million	200
2	C	population_in_million	120
3	A	gdp_percapita	2000
4	B	gdp_percapita	7000
5	C	gdp_percapita	15000

one column holds 2 variables

```
df2 = df.pivot(index = "countries")
```

table "df2"

each variable has its own column

countries	gdp_percapita	population_in_million
A	2000	100
B	7000	200
C	15000	120

common statistical fcns

- `count()` - # of non-null observations
- `sum()` - sum of values
- `mean()` - mean
- `median()` - median
- `std()` - standard deviation
- `var()` - variance
- `skew()` - skew (3rd moment)
- `kurt()` - kurtosis (4th moment)
- `quantile()` - sample quantile [value at %]
- `cov()` - covariance
- `corr()` - correlation

table styling

we can style a table by looking at individual values and applying sum style if a specific criteria is met:

```
def turn_number_red(val):
    """ takes a scalar and returns a
    a red string if scalar is negative;
    and black otherwise """
    colour = 'red' if val < 0 else 'black'
    return 'colour: %s' % colour
```

```
def highlight_max(column):
    """ highlight the largest number
    in a column """
    is_max = (column == column.max())
    return ['background-colour: yellow' if
            v else '' for v is_max]
```

```
df.style.apply(turn_number_red)
df.style.apply(highlight_max)
```

We can also format numbers to a specific decimal place

```
df.style.format("{:.2%}")
```

(`{:.2%}`: format # to 2 decimal places; add a percentage sign)

We can even add a caption to our table:

```
df.style.caption('the words we want to caption our table')
```

hide row indices:

```
df.style.hide_index()
```

Matplotlib (graphs and visualizations)

A visualization library used in python

The type of graph we plot depends on the number and type of variables.
What chart to plot and why:

# of categorical variables	# of quantitative variables	graph
1	0	bar graph
0	1	histogram
2	0	grouped bar graph
1	1	side-by-side boxplots
0	2	scatterplot
2	1	grouped boxplot
1	2	scatterplot with points identified by group