Python Quick
Reference

https://github.com/justmarkham/python-reference

By Kevin Markham (kevin@dataschool.io)

http://www.dataschool.io

Table of Contents:


```
'''


### IMPORTS ###

# 'generic import' of math module
import math
math.sqrt(25)

# import a function
from math import sqrt
sqrt(25) # no longer have to reference the module

# import multiple functions at once
from math import cos, floor

# import all functions in a module (generally discouraged)
from csv import *

# define an alias
```

```
import datetime as dt

# show all functions in math module
dir(math)

### DATA TYPES ###

# determine the type of an object
type(2) # returns 'int'
type(2.0) # returns 'float'
type('two') # returns 'str'
type(True) # returns 'bool'
type(None) # returns 'NoneType'

# check if an object is of a given type
isinstance(2.0, int) # returns False
isinstance(2.0, (int, float)) # returns True

# convert an object to a given type
float(2)
int(2.9)
str(2.9)

# zero, None, and empty containers are converted to False
bool(0)
bool(None)
bool('') # empty string
bool([]) # empty list
bool({}) # empty dictionary

# non-empty containers and non-zeros are converted to True
bool(2)
bool('two')
bool([2])

### MATH ###

# basic operations
10 + 4 # add (returns 14)
10 - 4 # subtract (returns 6)
10 * 4 # multiply (returns 40)
10 ** 4 # exponent (returns 10000)
5 % 4 # modulo (returns 1) - computes the remainder
```

```
10 / 4 # divide (returns 2 in Python 2, returns 2.5 in Python 3)
10 / float(4) # divide (returns 2.5)

# force '/' in Python 2 to perform 'true division' (unnecessary in Python 3)
from __future__ import division
10 / 4 # true division (returns 2.5)
10 // 4 # floor division (returns 2)

### COMPARISONS AND BOOLEAN OPERATIONS ###

# assignment statement
x = 5

# comparisons (these return True)
x > 3
x >= 3
x != 3
x == 5

# boolean operations (these return True)
5 > 3 and 6 > 3
5 > 3 or 5 < 3
not False
False or not False and True # evaluation order: not, and, or

### CONDITIONAL STATEMENTS ###

# if statement
if x > 0:
print('positive')

# if/else statement
if x > 0:
print('positive')
else:
print('zero or negative')

# if/elif/else statement
if x > 0:
print('positive')
elif x == 0:
print('zero')
else:
```

```
print('negative')

# single-line if statement (sometimes discouraged)
if x > 0: print('positive')

# single-line if/else statement (sometimes discouraged)
# known as a 'ternary operator'
'positive' if x > 0 else 'zero or negative'


### LISTS ###
## properties: ordered, iterable, mutable, can contain multiple data types

# create an empty list (two ways)
empty_list = []
empty_list = list()

# create a list
simpsons = ['homer', 'marge', 'bart']

# examine a list
simpsons[0] # print element 0 ('homer')
len(simpsons) # returns the length (3)

# modify a list (does not return the list)
simpsons.append('lisa') # append element to end
simpsons.extend(['itchy', 'scratchy']) # append multiple elements to end
simpsons.insert(0, 'maggie') # insert element at index 0 (shifts everything
right)
simpsons.remove('bart') # search for first instance and remove it
simpsons.pop(0) # remove element 0 and return it
del simpsons[0] # remove element 0 (does not return it)
simpsons[0] = 'krusty' # replace element 0

# concatenate lists (slower than 'extend' method)
neighbors = simpsons + ['ned', 'rod', 'todd']

# find elements in a list
simpsons.count('lisa') # counts the number of instances
simpsons.index('itchy') # returns index of first instance

# list slicing [start:end:step]
weekdays = ['mon', 'tues', 'wed', 'thurs', 'fri']
weekdays[0] # element 0
weekdays[0:3] # elements 0, 1, 2
```

```
weekdays[:3] # elements 0, 1, 2
weekdays[3:] # elements 3, 4
weekdays[-1] # last element (element 4)
weekdays[::2] # every 2nd element (0, 2, 4)
weekdays[::-1] # backwards (4, 3, 2, 1, 0)

# alternative method for returning the list backwards
list(reversed(weekdays))

# sort a list in place (modifies but does not return the list)
simpsons.sort()
simpsons.sort(reverse=True) # sort in reverse
simpsons.sort(key=len) # sort by a key

# return a sorted list (does not modify the original list)
sorted(simpsons)
sorted(simpsons, reverse=True)
sorted(simpsons, key=len)

# insert into an already sorted list, and keep it sorted
num = [10, 20, 40, 50]
from bisect import insort
insort(num, 30)

# create a second reference to the same list
same_num = num
same_num[0] = 0 # modifies both 'num' and 'same_num'

# copy a list (two ways)
new_num = num[:]
new_num = list(num)

# examine objects
num is same_num # returns True (checks whether they are the same object)
num is new_num # returns False
num == same_num # returns True (checks whether they have the same contents)
num == new_num # returns True


### TUPLES ###
## properties: ordered, iterable, immutable, can contain multiple data types
## like lists, but they don't change size

# create a tuple
```

```
digits = (0, 1, 'two') # create a tuple directly
digits = tuple([0, 1, 'two']) # create a tuple from a list
zero = (0,) # trailing comma is required to indicate it's a tuple

# examine a tuple
digits[2] # returns 'two'
len(digits) # returns 3
digits.count(0) # counts the number of instances of that value (1)
digits.index(1) # returns the index of the first instance of that value (1)

# elements of a tuple cannot be modified
digits[2] = 2 # throws an error

# concatenate tuples
digits = digits + (3, 4)

# create a single tuple with elements repeated (also works with lists)
(3, 4) * 2 # returns (3, 4, 3, 4)

# sort a list of tuples
tens = [(20, 60), (10, 40), (20, 30)]
sorted(tens) # sorts by first element in tuple, then second element
# returns [(10, 40), (20, 30), (20, 60)]

# tuple unpacking
bart = ('male', 10, 'simpson') # create a tuple
(sex, age, surname) = bart # assign three values at once


### STRINGS ###
## properties: iterable, immutable

# create a string
s = str(42) # convert another data type into a string
s = 'I like you'

# examine a string
s[0] # returns 'I'
len(s) # returns 10

# string slicing is like list slicing
s[:6] # returns 'I like'
s[7:] # returns 'you'
s[-1] # returns 'u'

# basic string methods (does not modify the original string)
```

```
s.lower() # returns 'i like you'
s.upper() # returns 'I LIKE YOU'
s.startswith('I') # returns True
s.endswith('you') # returns True
s.isdigit() # returns False (returns True if every character in the string is a digit)
s.find('like') # returns index of first occurrence (2), but doesn't support regex
s.find('hate') # returns -1 since not found
s.replace('like', 'love') # replaces all instances of 'like' with 'love'

# split a string into a list of substrings separated by a delimiter
s.split(' ') # returns ['I', 'like', 'you']
s.split() # equivalent (since space is the default delimiter)
s2 = 'a, an, the'
s2.split(',') # returns ['a', ' an', ' the']

# join a list of strings into one string using a delimiter
stooges = ['larry', 'curly', 'moe']
' '.join(stooges) # returns 'larry curly moe'

# concatenate strings
s3 = 'The meaning of life is'
s4 = '42'
s3 + ' ' + s4 # returns 'The meaning of life is 42'

# remove whitespace from start and end of a string
s5 = ' ham and cheese '
s5.strip() # returns 'ham and cheese'

# string substitutions: all of these return 'raining cats and dogs'
'raining %s and %s' % ('cats', 'dogs') # old way
'raining {} and {}'.format('cats', 'dogs') # new way
'raining {arg1} and {arg2}'.format(arg1='cats', arg2='dogs') # named arguments

# string formatting
# more examples: https://mkaz.tech/python-string-format.html
'pi is {:.2f}'.format(3.14159) # returns 'pi is 3.14'

# normal strings versus raw strings
print('first line\nsecond line') # normal strings allow for escaped characters
print(r'first line\nfirst line') # raw strings treat backslashes as literal characters
```

```
### DICTIONARIES ###
## properties: unordered, iterable, mutable, can contain multiple data types
## made of key-value pairs
## keys must be unique, and can be strings, numbers, or tuples
## values can be any type

# create an empty dictionary (two ways)
empty_dict = {}
empty_dict = dict()

# create a dictionary (two ways)
family = {'dad':'homer', 'mom':'marge', 'size':6}
family = dict(dad='homer', mom='marge', size=6)

# convert a list of tuples into a dictionary
list_of_tuples = [('dad', 'homer'), ('mom', 'marge'), ('size', 6)]
family = dict(list_of_tuples)

# examine a dictionary
family['dad'] # returns 'homer'
len(family) # returns 3
'mom' in family # returns True
'marge' in family # returns False (only checks keys)

# returns a list (Python 2) or an iterable view (Python 3)
family.keys() # keys: ['dad', 'mom', 'size']
family.values() # values: ['homer', 'marge', 6]
family.items() # key-value pairs: [('dad', 'homer'), ('mom', 'marge'), ('size',
6)]

# modify a dictionary (does not return the dictionary)
family['cat'] = 'snowball' # add a new entry
family['cat'] = 'snowball ii' # edit an existing entry
del family['cat'] # delete an entry
family['kids'] = ['bart', 'lisa'] # dictionary value can be a list
family.pop('dad') # remove an entry and return the value ('homer')
family.update({'baby':'maggie', 'grandpa':'abe'}) # add multiple entries

# access values more safely with 'get'
family['mom'] # returns 'marge'
family.get('mom') # equivalent
family['grandma'] # throws an error since the key does not exist
family.get('grandma') # returns None instead
```

```
family.get('grandma', 'not found') # returns 'not found' (the default)

# access a list element within a dictionary
family['kids'][0] # returns 'bart'
family['kids'].remove('lisa') # removes 'lisa'

# string substitution using a dictionary
'youngest child is %(baby)s' % family # returns 'youngest child is maggie'


### SETS ###
## properties: unordered, iterable, mutable, can contain multiple data types
## made of unique elements (strings, numbers, or tuples)
## like dictionaries, but with keys only (no values)

# create an empty set
empty_set = set()

# create a set
languages = {'python', 'r', 'java'} # create a set directly
snakes = set(['cobra', 'viper', 'python']) # create a set from a list

# examine a set
len(languages) # returns 3
'python' in languages # returns True

# set operations
languages & snakes # returns intersection: {'python'}
languages | snakes # returns union: {'cobra', 'r', 'java', 'viper', 'python'}
languages - snakes # returns set difference: {'r', 'java'}
snakes - languages # returns set difference: {'cobra', 'viper'}

# modify a set (does not return the set)
languages.add('sql') # add a new element
languages.add('r') # try to add an existing element (ignored, no error)
languages.remove('java') # remove an element
languages.remove('c') # try to remove a non-existing element (throws an error)
languages.discard('c') # remove an element if present, but ignored otherwise
languages.pop() # remove and return an arbitrary element
languages.clear() # remove all elements
languages.update(['go', 'spark']) # add multiple elements (can also pass a set)

# get a sorted list of unique elements from a list
sorted(set([9, 0, 2, 1, 0])) # returns [0, 1, 2, 9]
```

```
### DEFINING FUNCTIONS ###

# define a function with no arguments and no return values
def print_text():
print('this is text')

# call the function
print_text()

# define a function with one argument and no return values
def print_this(x):
print(x)

# call the function
print_this(3) # prints 3
n = print_this(3) # prints 3, but doesn't assign 3 to n
# because the function has no return statement

# define a function with one argument and one return value
def square_this(x):
return x**2

# include an optional docstring to describe the effect of a function
def square_this(x):
"""Return the square of a number."""
return x**2

# call the function
square_this(3) # prints 9
var = square_this(3) # assigns 9 to var, but does not print 9

# define a function with two 'positional arguments' (no default values) and
# one 'keyword argument' (has a default value)
def calc(a, b, op='add'):
if op == 'add':
return a + b
elif op == 'sub':
return a - b
else:
print('valid operations are add and sub')

# call the function
calc(10, 4, op='add') # returns 14
calc(10, 4, 'add') # also returns 14: unnamed arguments are inferred by position
```

```
calc(10, 4) # also returns 14: default for 'op' is 'add'
calc(10, 4, 'sub') # returns 6
calc(10, 4, 'div') # prints 'valid operations are add and sub'

# use 'pass' as a placeholder if you haven't written the function body
def stub():
pass

# return two values from a single function
def min_max(nums):
return min(nums), max(nums)

# return values can be assigned to a single variable as a tuple
nums = [1, 2, 3]
min_max_num = min_max(nums) # min_max_num = (1, 3)

# return values can be assigned into multiple variables using tuple unpacking
min_num, max_num = min_max(nums) # min_num = 1, max_num = 3


### ANONYMOUS (LAMBDA) FUNCTIONS ###
## primarily used to temporarily define a function for use by another function

# define a function the "usual" way
def squared(x):
return x**2

# define an identical function using lambda
squared = lambda x: x**2

# sort a list of strings by the last letter (without using lambda)
simpsons = ['homer', 'marge', 'bart']
def last_letter(word):
return word[-1]
sorted(simpsons, key=last_letter)

# sort a list of strings by the last letter (using lambda)
sorted(simpsons, key=lambda word: word[-1])


### FOR LOOPS AND WHILE LOOPS ###

# range returns a list of integers (Python 2) or a sequence (Python 3)
range(0, 3) # returns [0, 1, 2]: includes start value but excludes stop value
range(3) # equivalent: default start value is 0
range(0, 5, 2) # returns [0, 2, 4]: third argument is the step value
```

```
# Python 2 only: use xrange to create a sequence rather than a list (saves memory)
xrange(100, 100000, 5)

# for loop (not the recommended style)
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
print(fruits[i].upper())

# for loop (recommended style)
for fruit in fruits:
print(fruit.upper())

# iterate through two things at once (using tuple unpacking)
family = {'dad':'homer', 'mom':'marge', 'size':6}
for key, value in family.items():
print(key, value)

# use enumerate if you need to access the index value within the loop
for index, fruit in enumerate(fruits):
print(index, fruit)

# for/else loop
for fruit in fruits:
if fruit == 'banana':
print('Found the banana!')
break # exit the loop and skip the 'else' block
else:
# this block executes ONLY if the for loop completes without hitting 'break'
print("Can't find the banana")

# while loop
count = 0
while count < 5:
print('This will print 5 times')
count += 1 # equivalent to 'count = count + 1'

### COMPREHENSIONS ###

# for loop to create a list of cubes
nums = [1, 2, 3, 4, 5]
cubes = []
for num in nums:
```

```
cubes.append(num**3)

# equivalent list comprehension
cubes = [num**3 for num in nums] # [1, 8, 27, 64, 125]

# for loop to create a list of cubes of even numbers
cubes_of_even = []
for num in nums:
if num % 2 == 0:
cubes_of_even.append(num**3)

# equivalent list comprehension
# syntax: [expression for variable in iterable if condition]
cubes_of_even = [num**3 for num in nums if num % 2 == 0] # [8, 64]

# for loop to cube even numbers and square odd numbers
cubes_and_squares = []
for num in nums:
if num % 2 == 0:
cubes_and_squares.append(num**3)
else:
cubes_and_squares.append(num**2)

# equivalent list comprehension (using a ternary expression)
# syntax: [true_condition if condition else false_condition for variable in
iterable]
cubes_and_squares = [num**3 if num % 2 == 0 else num**2 for num in nums] # [1, 8,
9, 64, 25]

# for loop to flatten a 2d-matrix
matrix = [[1, 2], [3, 4]]
items = []
for row in matrix:
for item in row:
items.append(item)

# equivalent list comprehension
items = [item for row in matrix
for item in row] # [1, 2, 3, 4]

# set comprehension
fruits = ['apple', 'banana', 'cherry']
unique_lengths = {len(fruit) for fruit in fruits} # {5, 6}
```

```
# dictionary comprehension
fruit_lengths = {fruit:len(fruit) for fruit in fruits} # {'apple': 5, 'banana': 6, 'cherry': 6}
fruit_indices = {fruit:index for index, fruit in enumerate(fruits)} # {'apple': 0, 'banana': 1, 'cherry': 2}

### MAP AND FILTER ###

# 'map' applies a function to every element of a sequence
# ...and returns a list (Python 2) or iterator (Python 3)
simpsons = ['homer', 'marge', 'bart']
map(len, simpsons) # returns [5, 5, 4]
map(lambda word: word[-1], simpsons) # returns ['r', 'e', 't']

# equivalent list comprehensions
[len(word) for word in simpsons]
[word[-1] for word in simpsons]

# 'filter' returns a list (Python 2) or iterator (Python 3) containing
# ...the elements from a sequence for which a condition is True
nums = range(5)
filter(lambda x: x % 2 == 0, nums) # returns [0, 2, 4]

# equivalent list comprehension
[num for num in nums if num % 2 == 0]
```