Python Quick Reference by Data School (http://www.dataschool.io/)

Related: GitHub repository (https://github.com/justmarkham/python-reference) and blog post (http://www.dataschool.io/python-quick-reference/)

Table of contents

1. Imports

In [1]:
```
# 'generic import' of math module
import math
math.sqrt(25)
```

Out[1]:
```
5.0
```

In [2]:
```
# import a function
from math import sqrt
sqrt(25)    # no longer have to reference the module
```

Out[2]:
```
5.0
```

In [3]:
```
# import multiple functions at once
from math import cos, floor
```

In [4]:
```
# import all functions in a module (generally discouraged)
from csv import *
```

In [5]:
```
# define an alias
import datetime as dt
```

In [6]:
```
# show all functions in math module
print(dir(math))
```

Out[6]:
```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
```

[Back to top]

2. Data Types

Determine the type of an object:

In [7]:
```
type(2)
```

Out[7]:
```
int
```

In [8]:
```
type(2.0)
```

Out[8]:
```
float
```

In [9]:
```
type('two')
```
Out[9]:
```
str
```

In [10]:
```
type(True)
```
Out[10]:
```
bool
```

In [11]:
```
type(None)
```
Out[11]:
```
NoneType
```

Check if an object is of a given type:

In [12]:
```
isinstance(2.0, int)
```
Out[12]:
```
False
```

In [13]:
```
isinstance(2.0, (int, float))
```
Out[13]:
```
True
```

Convert an object to a given type:

In [14]:
```
float(2)
```
Out[14]:
```
2.0
```

In [15]:
```
int(2.9)
```
Out[15]:
```
2
```

In [16]:
```
str(2.9)
```
Out[16]:
```
'2.9'
```

Zero, None, and empty containers are converted to `False`:

In [17]:
```
bool(0)
```
Out[17]:
```
False
```

In [18]:
```
bool(None)
```
Out[18]:
```
False
```

In [19]:
```
bool('')    # empty string
```
Out[19]:
```
False
```

In [20]:
```
bool([])    # empty list
```
Out[20]:
```
False
```

In [21]:
```
bool({})    # empty dictionary
```
Out[21]:
```
False
```

Non-empty containers and non-zeros are converted to `True`:

In [22]:
```
bool(2)
```
Out[22]:
```
True
```

In [23]:
```
bool('two')
```
Out[23]:
```
True
```

In [24]:
```
bool([2])
```
Out[24]:
```
True
```

[Back to top]

3. Math

In [25]:
```
10 + 4
```
Out[25]: 14

In [26]:
```
10 - 4
```
Out[26]: 6

In [27]:
```
10 * 4
```
Out[27]: 40

In [28]:
```
10 ** 4    # exponent
```
Out[28]: 10000

In [29]:
```
5 % 4      # modulo - computes the remainder
```
Out[29]: 1

In [30]:
```
# Python 2: returns 2 (because both types are 'int')
# Python 3: returns 2.5
10 / 4
```
Out[30]: 2

In [31]:
```
10 / float(4)
```
Out[31]: 2.5

In [32]:
```
# force '/' in Python 2 to perform 'true division' (unnecessary in Python 3)
from __future__ import division
```

In [33]:
```
10 / 4     # true division
```
Out[33]: 2.5

In [34]:
```
10 // 4    # floor division
```
Out[34]: 2

[Back to top]

4. Comparisons and Boolean Operations

Assignment statement:

In [35]:
```
x = 5
```

Comparisons:

In [36]:
```
x > 3
```
Out[36]: True

In [37]:
```
x >= 3
```
Out[37]: True

In [38]:
```
x != 3
```
Out[38]: True

In [39]:
```
x == 5
```
Out[39]: True

Boolean operations:

In [40]:
```
5 > 3 and 6 > 3
```
Out[40]: True

In [41]:
```
5 > 3 or 5 < 3
```

Out[41]: True

In [42]:
```
not False
```

Out[42]: True

In [43]:
```
False or not False and True     # evaluation order: not, and, or
```

Out[43]: True

[Back to top]

5. Conditional Statements

In [44]:
```
# if statement
if x > 0:
    print('positive')
```

```
positive
```

In [45]:
```
# if/else statement
if x > 0:
    print('positive')
else:
    print('zero or negative')
```

```
positive
```

In [46]:
```
# if/elif/else statement
if x > 0:
    print('positive')
elif x == 0:
    print('zero')
else:
    print('negative')
```

```
positive
```

In [47]:
```
# single-line if statement (sometimes discouraged)
if x > 0: print('positive')
```

```
positive
```

In [48]:
```
# single-line if/else statement (sometimes discouraged), known as a 'ternary operator'
'positive' if x > 0 else 'zero or negative'
```

Out[48]: 'positive'

[Back to top]

6. Lists

- **List properties:** ordered, iterable, mutable, can contain multiple data types

In [49]:
```
# create an empty list (two ways)
empty_list = []
empty_list = list()
```

In [50]:
```
# create a list
simpsons = ['homer', 'marge', 'bart']
```

Examine a list:

In [51]:
```
# print element 0
simpsons[0]
```

Out[51]: 'homer'

In [52]:
```
len(simpsons)
```

Out[52]: 3

Modify a list (does not return the list):

In [53]:
```
# append element to end
simpsons.append('lisa')
simpsons
```

Out[53]:
```
['homer', 'marge', 'bart', 'lisa']
```

In [54]:
```
# append multiple elements to end
simpsons.extend(['itchy', 'scratchy'])
simpsons
```

Out[54]:
```
['homer', 'marge', 'bart', 'lisa', 'itchy', 'scratchy']
```

In [55]:
```
# insert element at index 0 (shifts everything right)
simpsons.insert(0, 'maggie')
simpsons
```

Out[55]:
```
['maggie', 'homer', 'marge', 'bart', 'lisa', 'itchy', 'scratchy']
```

In [56]:
```
# search for first instance and remove it
simpsons.remove('bart')
simpsons
```

Out[56]:
```
['maggie', 'homer', 'marge', 'lisa', 'itchy', 'scratchy']
```

In [57]:
```
# remove element 0 and return it
simpsons.pop(0)
```

Out[57]:
```
'maggie'
```

In [58]:
```
# remove element 0 (does not return it)
del simpsons[0]
simpsons
```

Out[58]:
```
['marge', 'lisa', 'itchy', 'scratchy']
```

In [59]:
```
# replace element 0
simpsons[0] = 'krusty'
simpsons
```

Out[59]:
```
['krusty', 'lisa', 'itchy', 'scratchy']
```

In [60]:
```
# concatenate lists (slower than 'extend' method)
neighbors = simpsons + ['ned', 'rod', 'todd']
neighbors
```

Out[60]:
```
['krusty', 'lisa', 'itchy', 'scratchy', 'ned', 'rod', 'todd']
```

Find elements in a list:

In [61]:
```
# counts the number of instances
simpsons.count('lisa')
```

Out[61]:
```
1
```

In [62]:
```
# returns index of first instance
simpsons.index('itchy')
```

Out[62]:
```
2
```

List slicing:

In [63]:
```
weekdays = ['mon', 'tues', 'wed', 'thurs', 'fri']
```

In [64]:
```
# element 0
weekdays[0]
```

Out[64]:
```
'mon'
```

In [65]:
```
# elements 0 (inclusive) to 3 (exclusive)
weekdays[0:3]
```

Out[65]:
```
['mon', 'tues', 'wed']
```

In [66]:
```
# starting point is implied to be 0
weekdays[:3]
```

Out[66]:
```
['mon', 'tues', 'wed']
```

In [67]:
```
# elements 3 (inclusive) through the end
weekdays[3:]
```

Out[67]:
```
['thurs', 'fri']
```

In [68]:
```
# last element
weekdays[-1]
```

Out[68]:
```
'fri'
```

In [69]:
```
# every 2nd element (step by 2)
weekdays[::2]
```

Out[69]:
```
['mon', 'wed', 'fri']
```

In [70]:
```
# backwards (step by -1)
weekdays[::-1]
```

Out[70]:
```
['fri', 'thurs', 'wed', 'tues', 'mon']
```

In [71]:
```
# alternative method for returning the list backwards
list(reversed(weekdays))
```

Out[71]:
```
['fri', 'thurs', 'wed', 'tues', 'mon']
```

Sort a list in place (modifies but does not return the list):

In [72]:
```
simpsons.sort()
simpsons
```

Out[72]:
```
['itchy', 'krusty', 'lisa', 'scratchy']
```

In [73]:
```
# sort in reverse
simpsons.sort(reverse=True)
simpsons
```

Out[73]:
```
['scratchy', 'lisa', 'krusty', 'itchy']
```

In [74]:
```
# sort by a key
simpsons.sort(key=len)
simpsons
```

Out[74]:
```
['lisa', 'itchy', 'krusty', 'scratchy']
```

Return a sorted list (does not modify the original list):

In [75]:
```
sorted(simpsons)
```

Out[75]:
```
['itchy', 'krusty', 'lisa', 'scratchy']
```

In [76]:
```
sorted(simpsons, reverse=True)
```

Out[76]:
```
['scratchy', 'lisa', 'krusty', 'itchy']
```

In [77]:
```
sorted(simpsons, key=len)
```

Out[77]:
```
['lisa', 'itchy', 'krusty', 'scratchy']
```

Insert into an already sorted list, and keep it sorted:

In [78]:
```
num = [10, 20, 40, 50]
from bisect import insort
insort(num, 30)
num
```

Out[78]:
```
[10, 20, 30, 40, 50]
```

Object references and copies:

In [79]:
```
# create a second reference to the same list
same_num = num
```

In [80]:
```
# modifies both 'num' and 'same_num'
same_num[0] = 0
print(num)
print(same_num)
```

```
[0, 20, 30, 40, 50]
[0, 20, 30, 40, 50]
```

In [81]:
```
# copy a list (two ways)
new_num = num[:]
new_num = list(num)
```

Examine objects:

In [82]:
```
num is same_num    # checks whether they are the same object
```

Out[82]:
```
True
```

In [83]:
```
num is new_num
```

Out[83]:
```
False
```

In [84]:
```
num == same_num    # checks whether they have the same contents
```

Out[84]:
```
True
```

In [85]:
```
num == new_num
```

Out[85]:
```
True
```

[Back to top]

7. Tuples

- **Tuple properties:** ordered, iterable, immutable, can contain multiple data types
- Like lists, but they don't change size

In [86]:
```
# create a tuple directly
digits = (0, 1, 'two')
```

In [87]:
```
# create a tuple from a list
digits = tuple([0, 1, 'two'])
```

In [88]:
```
# trailing comma is required to indicate it's a tuple
zero = (0,)
```

Examine a tuple:

In [89]:
```
digits[2]
```

Out[89]:
```
'two'
```

In [90]:
```
len(digits)
```

Out[90]:
```
3
```

In [91]:
```
# counts the number of instances of that value
digits.count(0)
```

Out[91]:
```
1
```

In [92]:
```
# returns the index of the first instance of that value
digits.index(1)
```

Out[92]:
```
1
```

Modify a tuple:

In [93]:
```
# elements of a tuple cannot be modified (this would throw an error)
# digits[2] = 2
```

In [94]:
```
# concatenate tuples
digits = digits + (3, 4)
digits
```

Out[94]:
```
(0, 1, 'two', 3, 4)
```

Other tuple operations:

In [95]:
```
# create a single tuple with elements repeated (also works with lists)
(3, 4) * 2
```

Out[95]:
```
(3, 4, 3, 4)
```

In [96]:
```
# sort a list of tuples
tens = [(20, 60), (10, 40), (20, 30)]
sorted(tens)    # sorts by first element in tuple, then second element
```

Out[96]:
```
[(10, 40), (20, 30), (20, 60)]
```

In [97]:
```
# tuple unpacking
bart = ('male', 10, 'simpson')    # create a tuple
(sex, age, surname) = bart        # assign three values at once
print(sex)
print(age)
print(surname)
```

```
male
10
simpson
```

[Back to top]

8. Strings

- **String properties:** iterable, immutable

In [98]:
```
# convert another data type into a string
s = str(42)
s
```

Out[98]:
```
'42'
```

In [99]:
```
# create a string directly
s = 'I like you'
```

Examine a string:

In [100]:
```
s[0]
```

Out[100]:
```
'I'
```

In [101]:
```
len(s)
```

Out[101]:
```
10
```

String slicing is like list slicing:

In [102]:
```
s[:6]
```

Out[102]:
```
'I like'
```

In [103]:
```
s[7:]
```

Out[103]:
```
'you'
```

In [104]:
```
s[-1]
```

Out[104]:
```
'u'
```

Basic string methods (does not modify the original string):

In [105]:
```
s.lower()
```

Out[105]:
```
'i like you'
```

In [106]:
```
s.upper()
```

Out[106]:
```
'I LIKE YOU'
```

In [107]:
```
s.startswith('I')
```

Out[107]:
```
True
```

In [108]:
```
s.endswith('you')
```

Out[108]:
```
True
```

In [109]:
```
# checks whether every character in the string is a digit
s.isdigit()
```

Out[109]:
```
False
```

In [110]:
```
# returns index of first occurrence, but doesn't support regex
s.find('like')
```

Out[110]:
```
2
```

In [111]:
```
# returns -1 since not found
s.find('hate')
```

Out[111]:
```
-1
```

In [112]:
```
# replaces all instances of 'like' with 'love'
s.replace('like', 'love')
```

Out[112]:
```
'I love you'
```

Split a string:

In [113]:
```
# split a string into a list of substrings separated by a delimiter
s.split(' ')
```

Out[113]:
```
['I', 'like', 'you']
```

In [114]:
```
# equivalent (since space is the default delimiter)
s.split()
```

Out[114]:
```
['I', 'like', 'you']
```

In [115]:
```
s2 = 'a, an, the'
s2.split(',')
```

Out[115]:
```
['a', ' an', ' the']
```

Join or concatenate strings:

In [116]:
```
# join a list of strings into one string using a delimiter
stooges = ['larry', 'curly', 'moe']
' '.join(stooges)
```

Out[116]:
```
'larry curly moe'
```

In [117]:
```
# concatenate strings
s3 = 'The meaning of life is'
s4 = '42'
s3 + ' ' + s4
```

Out[117]:
```
'The meaning of life is 42'
```

Remove whitespace from the start and end of a string:

In [118]:
```
s5 = '  ham and cheese  '
s5.strip()
```

Out[118]:
```
'ham and cheese'
```

String substitutions:

In [119]:
```
# old way
'raining %s and %s' % ('cats', 'dogs')
```

Out[119]:
```
'raining cats and dogs'
```

In [120]:
```
# new way
'raining {} and {}'.format('cats', 'dogs')
```

Out[120]:
```
'raining cats and dogs'
```

In [121]:
```
# new way (using named arguments)
'raining {arg1} and {arg2}'.format(arg1='cats', arg2='dogs')
```

Out[121]:
```
'raining cats and dogs'
```

String formatting (more examples (https://mkaz.tech/python-string-format.html)):

In [122]:
```
# use 2 decimal places
'pi is {:.2f}'.format(3.14159)
```

Out[122]:
```
'pi is 3.14'
```

Normal strings versus raw strings:

In [123]:
```
# normal strings allow for escaped characters
print('first line\nsecond line')
```

```
first line
second line
```

In [124]:
```
# raw strings treat backslashes as literal characters
print(r'first line\nfirst line')
```

```
first line\nfirst line
```

[Back to top]

9. Dictionaries

- **Dictionary 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

In [125]:
```
# create an empty dictionary (two ways)
empty_dict = {}
empty_dict = dict()
```

In [126]:
```
# create a dictionary (two ways)
family = {'dad':'homer', 'mom':'marge', 'size':6}
family = dict(dad='homer', mom='marge', size=6)
family
```

Out[126]:
```
{'dad': 'homer', 'mom': 'marge', 'size': 6}
```

In [127]:
```
# convert a list of tuples into a dictionary
list_of_tuples = [('dad', 'homer'), ('mom', 'marge'), ('size', 6)]
family = dict(list_of_tuples)
family
```

Out[127]:
```
{'dad': 'homer', 'mom': 'marge', 'size': 6}
```

Examine a dictionary:

In [128]:
```
# pass a key to return its value
family['dad']
```

Out[128]:
```
'homer'
```

In [129]:
```
# return the number of key-value pairs
len(family)
```

Out[129]:
```
3
```

In [130]:
```
# check if key exists in dictionary
'mom' in family
```

Out[130]:
```
True
```

In [131]:
```
# dictionary values are not checked
'marge' in family
```

Out[131]:
```
False
```

In [132]:
```
# returns a list of keys (Python 2) or an iterable view (Python 3)
family.keys()
```

Out[132]:
```
['dad', 'mom', 'size']
```

In [133]:
```
# returns a list of values (Python 2) or an iterable view (Python 3)
family.values()
```

Out[133]:
```
['homer', 'marge', 6]
```

In [134]:
```
# returns a list of key-value pairs (Python 2) or an iterable view (Python 3)
family.items()
```

Out[134]:
```
[('dad', 'homer'), ('mom', 'marge'), ('size', 6)]
```

Modify a dictionary (does not return the dictionary):

In [135]:
```
# add a new entry
family['cat'] = 'snowball'
family
```

Out[135]:
```
{'cat': 'snowball', 'dad': 'homer', 'mom': 'marge', 'size': 6}
```

In [136]:
```
# edit an existing entry
family['cat'] = 'snowball ii'
family
```

Out[136]:
```
{'cat': 'snowball ii', 'dad': 'homer', 'mom': 'marge', 'size': 6}
```

In [137]:
```
# delete an entry
del family['cat']
family
```

Out[137]:
```
{'dad': 'homer', 'mom': 'marge', 'size': 6}
```

In [138]:
```
# dictionary value can be a list
family['kids'] = ['bart', 'lisa']
family
```

Out[138]:
```
{'dad': 'homer', 'kids': ['bart', 'lisa'], 'mom': 'marge', 'size': 6}
```

In [139]:
```
# remove an entry and return the value
family.pop('dad')
```

Out[139]:
```
'homer'
```

In [140]:
```
# add multiple entries
family.update({'baby':'maggie', 'grandpa':'abe'})
family
```

Out[140]:
```
{'baby': 'maggie',
 'grandpa': 'abe',
 'kids': ['bart', 'lisa'],
 'mom': 'marge',
 'size': 6}
```

Access values more safely with `get`:

In [141]:
```
family['mom']
```

Out[141]:
```
'marge'
```

In [142]:
```
# equivalent to a dictionary lookup
family.get('mom')
```

Out[142]:
```
'marge'
```

In [143]:
```
# this would throw an error since the key does not exist
# family['grandma']
```

In [144]:
```
# return None if not found
family.get('grandma')
```

In [145]:
```
# provide a default return value if not found
family.get('grandma', 'not found')
```

Out[145]:
```
'not found'
```

Access a list element within a dictionary:

In [146]:
```
family['kids'][0]
```

Out[146]:
```
'bart'
```

In [147]:
```
family['kids'].remove('lisa')
family
```

Out[147]:
```
{'baby': 'maggie',
 'grandpa': 'abe',
 'kids': ['bart'],
 'mom': 'marge',
 'size': 6}
```

String substitution using a dictionary:

In [148]:
```
'youngest child is %(baby)s' % family
```

Out[148]:
```
'youngest child is maggie'
```

[Back to top]

10. Sets

- **Set 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)

In [149]:
```
# create an empty set
empty_set = set()
```

In [150]:
```
# create a set directly
languages = {'python', 'r', 'java'}
```

In [151]:
```
# create a set from a list
snakes = set(['cobra', 'viper', 'python'])
```

Examine a set:

In [152]:
```
len(languages)
```

Out[152]:
```
3
```

In [153]:
```
'python' in languages
```

Out[153]:
```
True
```

Set operations:

In [154]:
```
# intersection
languages & snakes
```

Out[154]:
```
{'python'}
```

In [155]:
```
# union
languages | snakes
```

Out[155]:
```
{'cobra', 'java', 'python', 'r', 'viper'}
```

In [156]:
```
# set difference
languages - snakes
```

Out[156]:
```
{'java', 'r'}
```

In [157]:
```
# set difference
snakes - languages
```

Out[157]:
```
{'cobra', 'viper'}
```

Modify a set (does not return the set):

In [158]:
```
# add a new element
languages.add('sql')
languages
```

Out[158]:
```
{'java', 'python', 'r', 'sql'}
```

In [159]:
```
# try to add an existing element (ignored, no error)
languages.add('r')
languages
```

Out[159]:
```
{'java', 'python', 'r', 'sql'}
```

In [160]:
```
# remove an element
languages.remove('java')
languages
```

Out[160]:
```
{'python', 'r', 'sql'}
```

In [161]:
```
# try to remove a non-existing element (this would throw an error)
# languages.remove('c')
```

In [162]:
```
# remove an element if present, but ignored otherwise
languages.discard('c')
languages
```

Out[162]:
```
{'python', 'r', 'sql'}
```

In [163]:
```
# remove and return an arbitrary element
languages.pop()
```

Out[163]:
```
'python'
```

In [164]:
```
# remove all elements
languages.clear()
languages
```

Out[164]:
```
set()
```

In [165]:
```
# add multiple elements (can also pass a set)
languages.update(['go', 'spark'])
languages
```

Out[165]:
```
{'go', 'spark'}
```

Get a sorted list of unique elements from a list:

In [166]:
```
sorted(set([9, 0, 2, 1, 0]))
```

Out[166]:
```
[0, 1, 2, 9]
```

[Back to top]

11. Defining Functions

Define a function with no arguments and no return values:

In [167]:
```
def print_text():
    print('this is text')
```

In [168]:
```
# call the function
print_text()
```

```
this is text
```

Define a function with one argument and no return values:

In [169]:
```
def print_this(x):
    print(x)
```

In [170]:
```
# call the function
print_this(3)
```

```
3
```

In [171]:
```
# prints 3, but doesn't assign 3 to n because the function has no return statement
n = print_this(3)
```

```
3
```

Define a function with one argument and one return value:

In [172]:
```
def square_this(x):
    return x**2
```

In [173]:
```
# include an optional docstring to describe the effect of a function
def square_this(x):
    """Return the square of a number."""
    return x**2
```

In [174]:
```
# call the function
square_this(3)
```

Out[174]:
```
9
```

In [175]:
```
# assigns 9 to var, but does not print 9
var = square_this(3)
```

Define a function with two 'positional arguments' (no default values) and one 'keyword argument' (has a default value):

In [176]:
```
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')
```

In [177]:
```
# call the function
calc(10, 4, op='add')
```

Out[177]:
```
14
```

In [178]:
```
# unnamed arguments are inferred by position
calc(10, 4, 'add')
```

Out[178]:
```
14
```

In [179]:
```
# default for 'op' is 'add'
calc(10, 4)
```

Out[179]:
```
14
```

In [180]:
```
calc(10, 4, 'sub')
```

Out[180]:
```
6
```

In [181]:
```
calc(10, 4, 'div')
```

```
valid operations are add and sub
```

Use `pass` as a placeholder if you haven't written the function body:

In [182]:
```
def stub():
    pass
```

Return two values from a single function:

In [183]:
```
def min_max(nums):
    return min(nums), max(nums)
```

In [184]:
```
# 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
```

Out[184]:
```
(1, 3)
```

In [185]:
```
# return values can be assigned into multiple variables using tuple unpacking
min_num, max_num = min_max(nums)
print(min_num)
print(max_num)
```

```
1
3
```

[Back to top]

12. Anonymous (Lambda) Functions

- Primarily used to temporarily define a function for use by another function

In [186]:
```
# define a function the "usual" way
def squared(x):
    return x**2
```

In [187]:
```
# define an identical function using lambda
squared = lambda x: x**2
```

Sort a list of strings by the last letter:

In [188]:
```
# without using lambda
simpsons = ['homer', 'marge', 'bart']
def last_letter(word):
    return word[-1]
sorted(simpsons, key=last_letter)
```

Out[188]:
```
['marge', 'homer', 'bart']
```

In [189]:
```
# using lambda
sorted(simpsons, key=lambda word: word[-1])
```

Out[189]:
```
['marge', 'homer', 'bart']
```

[Back to top]

13. For Loops and While Loops

range returns a list of integers (Python 2) or a sequence (Python 3):

In [190]:
```
# includes the start value but excludes the stop value
range(0, 3)
```

Out[190]:
```
[0, 1, 2]
```

In [191]:
```
# default start value is 0
range(3)
```

Out[191]:
```
[0, 1, 2]
```

In [192]:
```
# third argument is the step value
range(0, 5, 2)
```

Out[192]:
```
[0, 2, 4]
```

In [193]:
```
# Python 2 only: use xrange to create a sequence rather than a list (saves memory)
xrange(100, 100000, 5)
```

Out[193]:
```
xrange(100, 100000, 5)
```

for loops:

In [194]:
```
# not the recommended style
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
    print(fruits[i].upper())
```

```
APPLE
BANANA
CHERRY
```

In [195]:
```
# recommended style
for fruit in fruits:
    print(fruit.upper())
```

```
APPLE
BANANA
CHERRY
```

In [196]:
```
# 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)
```

```
('dad', 'homer')
('mom', 'marge')
('size', 6)
```

In [197]:
```
# use enumerate if you need to access the index value within the loop
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

```
(0, 'apple')
(1, 'banana')
(2, 'cherry')
```

for/else loop:

In [198]:
```
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")
```

```
Found the banana!
```

while loop:

In [199]:
```
count = 0
while count < 5:
    print('This will print 5 times')
    count += 1    # equivalent to 'count = count + 1'
```

```
This will print 5 times
This will print 5 times
This will print 5 times
This will print 5 times
This will print 5 times
```

[Back to top]

14. Comprehensions

List comprehension:

In [200]:
```
# for loop to create a list of cubes
nums = [1, 2, 3, 4, 5]
cubes = []
for num in nums:
    cubes.append(num**3)
cubes
```

Out[200]:
```
[1, 8, 27, 64, 125]
```

In [201]:
```
# equivalent list comprehension
cubes = [num**3 for num in nums]
cubes
```

Out[201]:
```
[1, 8, 27, 64, 125]
```

In [202]:
```
# 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)
cubes_of_even
```

Out[202]:
```
[8, 64]
```

In [203]:
```
# equivalent list comprehension
# syntax: [expression for variable in iterable if condition]
cubes_of_even = [num**3 for num in nums if num % 2 == 0]
cubes_of_even
```

Out[203]:
```
[8, 64]
```

In [204]:
```
# 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)
cubes_and_squares
```

Out[204]:
```
[1, 8, 9, 64, 25]
```

In [205]:
```
# 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]
cubes_and_squares
```

Out[205]:
```
[1, 8, 9, 64, 25]
```

In [206]:
```
# for loop to flatten a 2d-matrix
matrix = [[1, 2], [3, 4]]
items = []
for row in matrix:
    for item in row:
        items.append(item)
items
```

Out[206]:
```
[1, 2, 3, 4]
```

In [207]:
```
# equivalent list comprehension
items = [item for row in matrix
              for item in row]
items
```

Out[207]:
```
[1, 2, 3, 4]
```

Set comprehension:

In [208]:
```
fruits = ['apple', 'banana', 'cherry']
unique_lengths = {len(fruit) for fruit in fruits}
unique_lengths
```

Out[208]:
```
{5, 6}
```

Dictionary comprehension:

In [209]:
```
fruit_lengths = {fruit:len(fruit) for fruit in fruits}
fruit_lengths
```

Out[209]:
```
{'apple': 5, 'banana': 6, 'cherry': 6}
```

In [210]:
```
fruit_indices = {fruit:index for index, fruit in enumerate(fruits)}
fruit_indices
```

Out[210]:
```
{'apple': 0, 'banana': 1, 'cherry': 2}
```

[Back to top]

15. Map and Filter

`map` applies a function to every element of a sequence and returns a list (Python 2) or iterator (Python 3):

In [211]:
```
simpsons = ['homer', 'marge', 'bart']
map(len, simpsons)
```

Out[211]:
```
[5, 5, 4]
```

In [212]:
```
# equivalent list comprehension
[len(word) for word in simpsons]
```

Out[212]:
```
[5, 5, 4]
```

In [213]:
```
map(lambda word: word[-1], simpsons)
```

Out[213]:
```
['r', 'e', 't']
```

In [214]:
```
# equivalent list comprehension
[word[-1] for word in simpsons]
```

Out[214]:
```
['r', 'e', 't']
```

`filter` returns a list (Python 2) or iterator (Python 3) containing the elements from a sequence for which a condition is `True`:

In [215]:
```
nums = range(5)
filter(lambda x: x % 2 == 0, nums)
```

Out[215]:
```
[0, 2, 4]
```

In [216]:
```
# equivalent list comprehension
[num for num in nums if num % 2 == 0]
```

Out[216]:
```
[0, 2, 4]
```

[Back to top]