Python Cheat Sheet

Creator: Simon Zhang

rondom

```
import random
random.randint(0, 10) # [0, 10]
random.uniform(0, 10) # [0, 10], float
random.randrange(10) # [0, 10)
random.random() # [0, 1)

random.choice(['win', 'lose', 'draw'])      # Single random element from a
sequence

deck = 'ace two three four'.split()
random.shuffle(deck)                        # Shuffle a list
deck

random.sample([10, 20, 30, 40, 50], k=4)    # Four samples without
replacement
```

typing

```
from typing import List # type hint and check

def is_p(head: 'LinkedNode') -> List[int]:
    ...

# Generic concrete collections
class typing.Dict(dict, MutableMapping[KT, VT])
class typing.List(list, MutableSequence[T])
class typing.Set(set, MutableSet[T])

# These are both interpreted as get_names(employee_ids: List[Any]) ->
Dict[Any, Any]
def get_names(employee_ids: list) -> Dict:
  ...

def get_names(employee_ids: List) -> Dict:
  ...
```

style

3.5 Blank Lines Two blank lines between top-level definitions, be they function or class definitions. One blank line between method definitions and between the class line and the first method. No blank line following a def line. Use single blank lines as you judge appropriate within functions or methods.

No whitespace inside parentheses, brackets or braces.

```
Yes: spam(ham[1], {'eggs': 2}, [])
```

No whitespace before a comma, semicolon, or colon. Do use whitespace after a comma, semicolon, or colon, except at the end of the line.

```
Yes: if x == 4: print(x, y) x, y = y, x
```

No whitespace before the open paren/bracket that starts an argument list, indexing or slicing.

```
Yes: spam(1)
Yes: dict['key'] = list[index]
```

Surround binary operators with a single space on either side for assignment (=), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), and Booleans (and, or, not). Use your better judgment for the insertion of spaces around arithmetic operators (+, -, *, /, //, %, **, @).

Never use spaces around = when passing keyword arguments or defining a default parameter value, with one exception: when a type annotation is present, do use spaces around the = for the default parameter value.

```
Yes: def complex(real, imag=0.0): return Magic(r=real, i=imag)
Yes: def complex(real, imag: float = 0.0): return Magic(r=real, i=imag)
```

Don't use spaces to vertically align tokens on consecutive lines, since it becomes a maintenance burden (applies to :, #, =, etc.):

```
foo = 1000  # comment
long_name = 2  # comment that should not be aligned

dictionary = {
    'foo': 1,
    'long_name': 2,
}

# import separate line
import os
import sys
from typing import Mapping, Sequence
import tensorflow as tf


# docstring
"""A one line summary of the module or program, terminated by a period.

Leave one blank line.  The rest of this docstring should contain an
overall description of the module or program.  Optionally, it may also
contain a brief description of exported classes and functions and/or usage
examples.
```

```
  Typical usage example:

  foo = ClassFoo()
  bar = foo.FunctionBar()
"""


def fetch_smalltable_rows(table_handle: smalltable.Table,
                          keys: Sequence[Union[bytes, str]],
                          require_all_keys: bool = False,
) -> Mapping[bytes, Tuple[str]]:
    """Fetches rows from a Smalltable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by table_handle.  String keys will be UTF-8 encoded.

    Args:
        table_handle: An open smalltable.Table instance.
        keys: A sequence of strings representing the key of each table
          row to fetch.  String keys will be UTF-8 encoded.
        require_all_keys: Optional; If require_all_keys is True only
          rows with values set for all keys will be returned.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {b'Serak': ('Rigel VII', 'Preparer'),
         b'Zim': ('Irk', 'Invader'),
         b'Lrrr': ('Omicron Persei 8', 'Emperor')}

        Returned keys are always bytes.  If a key from the keys argument
is
        missing from the dictionary, then that row was not found in the
        table (and require_all_keys must have been False).

    Raises:
        IOError: An error occurred accessing the smalltable.
    """

    # start method body...

class SampleClass:
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """
```

```
    def __init__(self, likes_spam: bool = False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_var_name, instance_var_name, function_parameter_name, local_var_name.

```
# conditional expression
one_line = 'yes' if predicate(value) else 'no'
slightly_split = ('yes' if predicate(value)
                  else 'no, nein, nyet')
the_longest_ternary_style_that_can_be_done = (
    'yes, true, affirmative, confirmed, correct'
    if predicate(value)
    else 'no, false, negative, nay')
```

library

```
# keep list sorted after insertion
bisect.bisect([1, 2, 3, 4, 5], target=3)
'''
binary search, return insertion index of first element larger than target
The returned insertion point i partitions the array a into two halves so
that
all(val <= x for val in a[lo:i]) for the left side and
all(val > x for val in a[i:hi]) for the right side.
'''

bisect.bisect_left(a, x, lo=0, hi=len(a))
# return before (to the left of) any existing entries
# all(val < x for val in a[lo:i]) for the left side and
# all(val >= x for val in a[i:hi]) for the right side.
```

str <-> int, split/join

```
str(10)
int('10', base=10)

'this is my string'.split()
strings = ['1', '2', '3', 'do', 're', 'mi']
','.join(strings)
```

comparator with more item use tuple (most, ..., least)

```
l1 = [(1, 10), (2, 100), (2, -5), (1, -100)]
l1.sort(key = lambda x: (x[0], -x[1])) # - means decreasing
# overwrite __gt__, __lt__
print(l1)
```

sortedcontainers

```
from sortedcontainers import SortedList
sl = SortedList(['e', 'a', 'c', 'd', 'b'])
sl
SortedList(['a', 'b', 'c', 'd', 'e'])
sl *= 10_000_000
sl.count('c')
10000000
sl[-3:]
['e', 'e', 'e']

from sortedcontainers import SortedDict
sd = SortedDict({'c': 3, 'a': 1, 'b': 2})
sd
SortedDict({'a': 1, 'b': 2, 'c': 3})
sd.popitem(index=-1)
('c', 3)

from sortedcontainers import SortedSet
ss = SortedSet('abracadabra')
ss
SortedSet(['a', 'b', 'c', 'd', 'r'])
ss.bisect_left('c')
2
```

deque

```
dq = collections.deque([])
dp.appendleft(1)
dp.popleft()

data = '1,2,#,#,3,4,#,#,5,#,#'
vals = collections.deque(val for val in data.split(','))
```

main / test

```
# when import, will NOT run the code under 'main', only run top-level code
def foo():
    ...

if __name__ == '__main__':
    foo()

    sol = Solution()
    print(sol.func_name(args))
```

lambda

```
# lambda
lambda x : x
(lambda x : x + 1)(2) # (func)(args)
(lambda x, y: x + y)(2, 3)

high_ord_func = lambda x, func: x + func(x)
high_ord_func(3, lambda x: x ** 2)
sorted(name, key=lambda x: x.age) # sort by age ascending
```

XX.sort vs sorted()

```
digits = [0, 5, 7, 3, 4, 9, 1, 6, 3, 8]

digits.sort(reverse=True) # change input, in-place
print(digits) # [9, 8, 7, 6, 5, 4, 3, 3, 1, 0]

digits = [0, 5, 7, 3, 4, 9, 1, 6, 3, 8]
sorted(digits, reverse=True) # return copy of input, not change input, not
in-place
print(digits) # [0, 5, 7, 3, 4, 9, 1, 6, 3, 8]
```

Counter

```
from collections import Counter, defaultdict

word = "mississippi"
counter = defaultdict(int)
for letter in word:
    counter[letter] += 1

# Use a string as an argument
letters = Counter('mississippi')
Counter(set("mississippi")) # Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

```
letters.update("missouri") # Counter({'i': 6, 's': 6, 'p': 2, 'm': 2, 'o':
1, 'u': 1, 'r': 1})

for letter, freq in letters.items():
    print(letter, freq)
print(letters.keys()) # dict_keys(['m', 'i', 's', 'p'])
print(letters.values()) # dict_values([1, 4, 4, 2])

counter1.most_common(3) # default sort and return all
counter1.subtract(counter2) # c1 = c1 - c2
counter1.update(counter2) # c1 = c1 + c2
c1 +/-/&/| c2
# return positve, & == min, | == max
```

List -> list() / [1, 2]

```
l1 = [1, 2, 3]
l2 = [4, 5]
l1.extend(l2)
l1.append(100)
l1.insert(0, 666)

l1.pop(idx) # default pop last element, -1 means last element
l1.remove(2) # remove first accurance (value)
del l1[1] # remove at specific index

l1.reverse() # change in place, no return
l1.sort(min) # decresing order
l1.index(4) # first accurance
l1.count(0)
l1.sort(key=None, reverse=False) # change l1
sorted(l1) # l1 not change

# 2D matrix
arr3 = [[0 for _ in range(1)] for _ in range(5)]
arr3[1][0] = 9
arr3 # [[0], [9], [0], [0], [0]]
```

String -> str

```
s = 'hello simon'
s.title() # 'Hello Simon'
len(s)
s[5:10]
s[-3:-1]
s.upper()
s.lower()
s.index('h')
s[0] == 'h'
```

```
s = '    Haha,   SSSimonnnnn   '
s.strip()
s.strip().split(',')
s.count('S')
s.find('XX') # -1 if not find
s.index('S') # throw error if not find

# like Java StringBuilder
l = []
l.append('foo')
l.append('bar')
l.append('baz')
s = ''.join(l)

int('123')
float('1.22')

# f-strings, Formatted string literal
import math
print(f'The value of pi is approximately "{math.pi:.3f}"')

table = {'Simon': 178, 'helen': 165, 'dudu': 30}
for name, age in table.items():
    print(f'{name:10} ==> {age:10}')

print('{1} and {0}'.format('one', 'two')) # position
print('This {food} is {adjective}.'.format(food='KFC', adjective='Bad')) #
keyword

table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
      'Dcab: {0[Dcab]:d}'.format(table))
```

Dictionary -> dict() / {'one' : 1} / {'one'=1}

```
name = {'Simom': 178,
        'Helen': 165}
name['Simom']
name['Helen'] = 999
name['xxx'] # KeyError
name.get('xxx') # return None
name.get('Dudu', 30) # default value is 30
name.setdefault('Dudu', 30) # only set when key is not in dict

for key in name: # same as keys()
    print(key)

name['GuaiGuai'] = 25 # add / modify dict use dict[key] = value
del name['GuaiGuai']
name.pop('Helen', default=-1) # return value, remove entry
name.popitem() # return 2-tuple, LIFO order
```

```
name.keys()
name.values() # max(d.values()) return max value
name.items()

# f-string, concatenate string and other type
for key, value in name.items():
    print(f'{key} height: {value}')
```

defaultdict

```
name_default = defaultdict(list)
name_default[0] # return [], no Error

# list for append
s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
d = defaultdict(list)
for k, v in s:
    d[k].append(v)

sorted(d.items())
# [('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]


# int for count
s = 'mississippi'
d = defaultdict(int)
for k in s:
    d[k] += 1

sorted(d.items())
# [('i', 4), ('m', 1), ('p', 2), ('s', 4)]


# set for de-duplicate
s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1),
('blue', 4)]
d = defaultdict(set)
for k, v in s:
    d[k].add(v)

sorted(d.items())
# [('blue', {2, 4}), ('red', {1, 3})]
```

Tuple -> tuple() / (1,)

```
t1 = 1,
t2 = (1, 2, 3, 100, 5, 6)
t3 = tuple('hello') # ('h', 'e', 'l', 'l', 'o')
```

```
for t in t3:
    print(t)
max(t2)
sorted(t2)
len(t3)
```

Set -> set()

```
s = {c for c in 'abracaabroa' if c not in 'abc'} # {'r', 'o'}
s2 = set('foobar') # {'r', 'f', 'b', 'a', 'o'}
s3 = set(['a', 'b', 'foo'])

s.add(x)
s.remove(x) # raise error if absence
s.discard(x) # return None if absence

# de-duplicate
t = [1, 2, 3, 1, 2, 5, 6, 7, 8]
list(set(t))
# [1, 2, 3, 5, 6, 7, 8]

# de-duplicate + ordered!
list(dict.fromkeys('abracadabra'))
# ['a', 'b', 'r', 'c', 'd']

s.copy() # deep copy
s.update(s2) s == s2

s.isdisjoint(s2)
s.issubset(s2)
s < s2
s.union(s2) == s | s2
s.intersection(s2) == s & s2
s2.difference(s) == s2 - s
s.symmetric_difference(s2) == s ^ s2 # either in s or s2, but not both
```

zip / enumerate

```
key = 'abc'
value = '123'
for pair in zip(key, value):
    print(pair)

list(enumerate('simon')) # [(0, 's'), (1, 'i'), (2, 'm'), (3, 'o'), (4,
'n')]
list(zip(range(5), 'simon'))
d = dict(enumerate('abc')) # {0: 'a', 1: 'b', 2: 'c'}
z = zip('xyz', [23, 24, 25])
```

```
dict(z)
d1 = {(0, 0) : 5}
```

stack

```
stack = list(range(6))
stack.append(10)
stack.append(20)
stack.pop()
```

queue

```
from collections import deque
queue = deque(['simon', 'helen', 'dudu'])
queue.append('neinei')
queue.popleft()
queue.popleft()
queue[0] # peek
queue.clear()
```

Priority Queue

```
from heapq import heappush, heappop, heapify

def heapsort(iterable):
    h = []
    for value in iterable:
        heappush(h, value)
    return [heappop(h) for _ in range(len(h))]

l = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
print(heapsort(l))

heap[0] is smallest
heapq.heappush(heap, item)
heapq.heappop(heap) # return smallest
heapq.heappushpop(heap, item) # == push + pop
heap[0] # peek smallest
heapq.heapify(list())
heap.heapreplace(heap, item) # == pop + push
heapq.merge(*iterables, key=None, reverse=False)
heapq.nlargest(n, iterable, key=None)
```

math

```
-float('inf') == -math.inf
fabs(-100) # absolute value
math.pow(2,10) # 2 ^ 10 = 1024
math.log(math.e) # ln(e) = 1
math.log(math.e ** 2,math.e) # ln(e^2) / ln(e) = 2
```

ASCII

```
print(ord('a'))    # 97
print(ord('A'))    # 65
print(ord('0'))    # 48

print(chr(97)) # 'a'
print(chr(65))  # 'A'
print(chr(48)) # '0'
```

Comprehensions & Generator

```
result = [mapping_expr for value in iterable if filter_expr]

result = [{'key': value} for value in iterable
        if a_long_filter_expression(value)]

result = [complicated_transform(x)
        for x in iterable if predicate(x)]

descriptive_name = [
    transform({'key': key, 'value': value}, color='black')
    for key, value in generate_iterable(some_input)
    if complicated_condition_is_met(key, value)
]

result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))

return {x: complicated_transform(x)
        for x in long_generator_function(parameter)
        if x is not None}

squares_generator = (x**2 for x in range(10))

unique_names = {user.name for user in users if user is not None}

eat(jelly_bean for jelly_bean in jelly_beans
    if jelly_bean.color == 'black')
```

class

```
class Student:
    """This is a student class."""

    def __init__(self, name = "") -> None:
        self.name = name

    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def speak(self):
        print(f'Hello, World! I am {self.name}')

    def __private(self): # private
        print("NO")

    def public(self):
        print("YES")
        self.__private()

    def __repr__(self) -> str:
        return f'student name: {self.name}'

    def __str__(self) -> str:
        return '“informal” or nicely printable string representation of
object.'

stu = Student()
stu.set_name("Simon")
print(stu.__dict__) # dict of attributes
print(stu) # __str__
print(repr(stu)) # __repr__
print(stu.__doc__) # return """doc"""
```

property

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x
```

```
    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

c = C()
c.x = 10 # setx
print(c.x) # getx
del c.x # delx

# class property([fget[, fset[, fdel[, doc]]]])
# x = property(getx, setx, delx, "I'm the 'x' property.")
```

decorator

```
def timer(fun):
    def inner():
        <inner_body>
        fun()
    return inner

import time

# no param
def timer(func):
    def inner():
        start = time.time()
        func()
        end = time.time()
        print('function {} cost {} seconds'.format(func.__name__,
round(end - start, 2)))
    return inner

# with params
def repeat_func(n):
    def wrapper(func):
        def inner():
            print('before function run')
            for i in range(n):
                func()
            print('after function run')
        return inner
    return wrapper
```