**# Python Formula Sheet**

**Bools**

# If and not convert to bool

bool(lst) # len(lst) != 0

not [ ] # True

bool(num) # num != 0

True+True # 2

# Implicit conversion 0:False,1:True

**Scope**

def f(lst):

lst[0] = 1 # happens

lst = [1, 2, 3] # does not happen

lst = [0, 2]

f(lst)

print(lst) # [1,2]

**List Comprehension**

lst = [4, 1, 2, 3]

lst[0] = 0  # lists are mutable

[el ^ 2 for el in lst]  # apply

[el for el in lst if el % 2 == 0] # filter

lst + [4, 5] # concatenate

lst1 = [1, 2, 3]; lst2 = [2, 3, 4]

[(x, y) for x in lst1 for y in lst2] #AxB

**List Functions**

sum(lst) # sum(iter,start=0)

any(el == 2 for el in lst)

all(el % 2 == 0 for el in lst)

mat = [[1, 2, 4], [0, 2, 3]]

sorted(mat, *key*=len, *reverse*=True) # key is the comparison

min(mat, *key*=min)  # [0,2,3]

max(mat, *key*=max) # [1,2,4]

# returns the element not key(el)

**Iterables and Immutables**

range(1, 8, 2)  # w/out stop

# range(start,stop,step)

zip(lst, lst[1:])  # iter of tpl (el1,el2) # w/ same index

enumerate(lst)

# iter of tuple (index,el)

*set*([1, 2, 2, 3, 3])  # {1,2,3}

# no repeats and no order

map(len, mat) # iter of func(el)

filter(lambda x: min(x) > 0, mat)

# iter if cond(el)

**String**

"aab".upper() # “AAB”

"a-ba-c".replace("-",";") # "a;ba;c"

";".join(["a", "b", "c"]) # "a;b;c"

# joined [str] by separator

"a,ba,c".split(",") # ["a", "ba", "c"]

"aab".count("a") # 2

"baab".find("a") # 1

# -1 if not found

f"list: {', '.join(map(str,lst))}"

# formatting

**Tuples**

tpl = (1, 2, 3) # Create Tuple

a, b = (1, 2) # unpacking

**Slicing**

string = "abcdab"

string[-1] # last element

# string[start:stop:step]

# stop not including

string[1:] # tail

string[:-1] # init

string[::-1] # reverse

**Dictionary**

dictn = {"a": 2, "b": 3, "c": -1}

dictn["a"] # access/alter key

dictn.items() # iter of tpls (key, val)

dictn.values() # iter of values

dictn.keys() #(or dictn) iter of keys

{key: value \* 2 for key, value in dictn.items() if value != 0}

dict(zip(["a", "b", "c"], [2, 3, 1]))

dictn.get("d", 0) # .get(key,default)

"a" in dictn # True

**Recursion**

Standard

def change\_rec(n, lst):

return (n== 0 if n <= 0 else 0 if not lst else change\_rec(n - lst[0], lst) + change\_rec(n, lst[1:]))

Memoized

def change\_mem(n, lst, memo={}):

if n <= 0: return n == 0

elif not lst: return 0

key = (n, len(lst))

if key not in memo:

memo[key] = change\_mem(n - lst[0], lst, memo) + change\_mem(n, lst[1:], memo)

return memo[key]

Accumulator (Fold)

def prod(lst, acc=1):

return (acc if not lst

else prod(lst[1:], acc \* lst[0]))

**Algorithms**

# Iterate a product

prodLst = 1

for el in lst:

prodLst \*= el

# Iterate a Max

max\_ = lst[0]

for el in lst:

max\_ = el if el > max\_ else max\_

# MultiMax Index

max\_ = max(lst)

indexes=[i for i, e in enumerate(lst)

if e == max\_]

# Make a flat

flat = lambda mat: [el for lst in mat for el in lst]

# Sum a flat

flat = lambda lst: sum(lst, [])

# Histogram

hist = lambda s: {char: s.count(char) for char in s}

# Indexing

def indexes(string):

h = {}

for i, char in enumerate(string):

h[char] = h.get(char, []) + [i]

return h

# Hist of Longest Sequence

hist\_longest = lambda s: { char: max(k for k in range(len(s))

if char \* k in s) for char in s}

# Filter Sparse

sparse = lambda dictn: dict(filter(lambda t: t[1] != 0, dictn.items()))

**Classes**

class Struct:

def \_\_init\_\_(self, p1, p2, p3):

if not (type(p2) is str):

raise ValueError("")

self.p1, self.p2= (p1, p2)

self.p3 = p3

def \_\_repr\_\_(self):

return "\n".join(map(": ".join,[

("p1", str(self.p1)),

("p2", self.p2),

("p3", ", ".join(

f"{a}: {' '.join(map(str, b))}"

for a, b in self.p3.items()

)),

],))

def func1(self, p):

return self.p1 + p

**Subclasses**

class SubStruct(Struct):

def \_\_init\_\_(self, p1, p2, p3, p4):

Struct.\_\_init\_\_(self, p1, p2, p3)

self.p4 = p4

def \_\_repr\_\_(self):

return Struct.\_\_repr\_\_(self) + f"\np4: {self.p4}"

def func2(self, p):

return self.p4 - p

data = SubStruct(2,"a", {"a":(2,3),"b": (3,1)},3)

data.func1(3) # 5 # Inherited

data.func2(4) # -1

**IO**

def read\_table(filename):

try:

with open(filename) as f:

return [row.split(",") for row in f.read().split("\n")]

except IOError:

raise IOError("Error!")

return [] # either

**Numpy**

import numpy as np

data = np.array([[1, 2, 3, 4], [2, 3, 1, 5], [2, 6, 0, 0]], dtype=int)

data.shape # (3,4)

np.zeros((2, 3), dtype=int)

np.ones((2, 3), dtype=int)

np.arange(10) #np.array(range(10))

data[1, 2] # 1

# indexing w/ commas row 1, col 2

data[1:3, :3] # array([[1, 2],[2, 3]])

# slicing

data \* 2 # element wise operation

# axis=1 is columns, axis=0 is rows

data.sum() # total sum

data.sum(axis=1) # row sum

# sum by axis 1

data.sum(axis=0) # column sum

np.argmax(data.sum(axis=0))

np.hstack([np.zeros((3, 1)), data])

np.vstack([np.zeros((1, 4)), data])

np.diff(data) # side difference

data[data > 2] # Masks

(data>2).sum() # Total w/ condition

np.where(data>3)

# (array([0, 1, 2]), array([3, 3, 1]))

np.where(data>3,data,-1)

# return data sub -1 where data<=3

np.unique(data) # np.arange(7)

**Pandas**

def read\_pd(filename, index\_col):

try:

return pd.read\_csv(filename, index\_col=index\_col)

except IOError:

raise IOError("Error!")

return pd.DataFrame() # either

# pd.read\_csv(filepath, sep=',', header='infer', names=None, index\_col=None, usecols=None, dtype=None, comment=None, encoding=None)

df = pd.DataFrame(data={"a": [3, 1, 4, 5], "b": [1, 2, 8, 1], "c": ["w", "x", "y", "z"]})

df["a"] # gets Series of that column

df[["a", "b"]] # gets DF of those cols

df.max(axis=0) # Max of each col

df.max().max()

df[["a", "b"]].apply(lambda row: row.min(), axis=1)

df.mean(axis=1, numeric\_only=True) # row mean

df[df["a"] > 3] # Mask

df.loc[df["a"] > 3, "b"]

# Find row in mask and col is b

df = df.drop("c", axis=1) # drops col

df[df > 1].count() # counts the non-NaN > 1

df[~df.isin([1,2])].count()

i = df["a"].idxmax() # index of row of max value

df.loc[i, "a"] # locates the element

df.iloc[lambda x: x.index % 2 == 0]

# gets the even indexed rows

df.append(df.mean(axis=0, numeric\_only=True), ignore\_index=True)

df.groupby(["b"]).mean()

df1 = pd.DataFrame({"A": ["a", "b", "a"], "B": [1, 2, 4]})

df1["name"] = "One"

df2 = pd.DataFrame({"A": ["c", "b", "a"], "B": [2, 3, 0]})

df2["name"] = "Two"

pd.concat([df1, df2], ignore\_index=True)

# pd.concat(dfs, axis=0, join='outer', ignore\_index=False)

**ImageIO**

import imageio

def compute\_entropy(img):

im = imageio.imread(img).flatten() # open

h, bins = np.histogram(im, bins=list(range(255)), density=True)

return np.sum(-h \* np.log2(h, where=(h != 0)))

def nearest\_enlarge(img, a):

im = imageio.imread(img)

return np.array([

[im[i // a, j // a] for j in range(im.shape[1] \* a)]

for i in range(im.shape[0] \* a)

])

segmentation = lambda im,thr: (im>thr)\*255

def neig(im, x, y, dx=2, dy=2):

xL, xR = (max(x - dx, 0), min(x + dx + 1, im.shape[0]))

yB, yT = (max(y - dy, 0), min(y + dy + 1, im.shape[1]))

return im[xL:xR, yB:yT]

def morph\_by\_neig(im, func, dx=2, dy=2):

return np.array([

[func(neig(im, x, y, dx, dy)) for y in range(im.shape[1])]

for x in range(im.shape[0])

])

def erosion(im, dx=2, dy=2):

return morph\_by\_neig(im, np.min, dx, dy)