**Range ICS 31 FINAL Cheetsheet**

For x in range(20, -1,-1)

(20 start point, mid(-1) = end point but not including itself, interval)

**Dictionary**

# Constant dictionaries

d = { } # Curly braces; this is an empty dict

# Non-empty dict has key-value pairs:

d = {"ICS 31": Course("ICS", "31", "kay", 4.0),

"ICS 32": Course("ICS", "32", "thornton", 4.0)}

We look things up in a dict using a KEY D[k] returns corresp. value

-- Keys must be an immutable type (number, string, tuple; not list)

-- Values can be anything (lists, tuples, whatever)

-- Keys must be UNIQUE. Can't have two entries for 'Joe'

-- Key-value pairs are NOT ORDERED. (But you can extract all the

info into a list and then sort that list.)

Indexing: d['Donald Duck'] value is whatever matches, the value that corresponds to that key.

d.pop() removes value from dictionary

**TUPLE** -- Packaging up multiple parts But no type name, no field/attribute names

Like namedtuples, except:

-- But no type name, no field/attribute names

-- Tuples are built in to Python; no importing.

Use tuples for:

-- Packaging up multiple parts, e.g., to return from a function.

Useful where package is small, maybe 2-3 fields.

Otherwise use namedtuple so you don't have to remember the order

-- Use as multipart key in a dict

**SET:** Like a mathematical set.

-- No duplicate entries: All elements unique

-- Order doesn't matter.

s = {1, 2, 3, 4} # Set has no colons.

d = {1: 'a', 2: 'b'} # Dict has key-value pairs with colon

ed = { } # Empty dictionary

print(type(ed))

es = set() # set constructor with no arguments -> **empty set**

**Set operations:**

in not in len(s)

Set comparison:

s == r s != r s < r (strict subset) s <= r > >=

Set operators:

**union** (combine 2 sets) s | r **intersection** (common elements) s & r

**subtraction** (removing members) s - r **symmetric** difference (in one but not both):s ^ r

Set methods:

**s.add(i)**: Insert i into s (if not already there) **s.remove(i)**: Remove item, error if not there

**s.discard(i)**: Remove item, shut up if not there **s.clear()**: Remove all the items

The **keys()** method gives a view of all the keys in the dict; **list()** turns the view into a list.

**READ**

1. **read()** -- a string containing the entire file (including newlines) ‘\nThere is a blank line above this line.\n’ Would be more effective if the input file were organized into fixed-width columns.

2. **read(n)** -- a string containing n characters, the next n characters in the input *infile.read(1) ->* ‘T’ *infile.read(5) ->* ‘he 3 ’

3. **readline()** -- a string containing the (rest of the) next line to be read *‘lines in tis file end with the new line character.\n’*

4. **readlines()** -- a list containing each line in the file

5. **for line in file** -- the control variable (line) is a string that

        holds the next line in the fie, each time the loop repeats.

A **mapping** operation does something to each item in the list, returning a list of changed items. Changing all the prices in a restaurant is a mapping operation.  Doubling each number on a list of numbers is another.

A **filtering** operation selects (just some) items from a list.  Collecting just the odd numbers from a list of numbers, or just the Thai restaurants from a list of restaurants, is a filtering operation.

A **reducing** operation (also called folding or accumulating) takes a list and combines it into one single value, like adding up all the numbers on the list.

**Find(), replace(), maketrans(), translate()**

def redact term (name: str) -> str:

''' Return the name with each letter replaced with X or x (according to its original upper or lower case) and other characters ‘’’

alphabet = 'abcdefghijklmnopqrstuvwxyz' x\_string = 'xxxxxxxxxxxxxxxxxxxxxxxxxx'

ALPHABET = alphabet.upper() X\_STR = x\_string.upper()

table = str.maketrans(alphabet+ALPHABET, x\_string+X\_STR) **str.maketrans(x,y)-> dict Return translation table usable for translate**

return name.translate(table) **x.translate(table)-> str** Return copy of string x where all characters have been mapped through the given translation table. Unmapped charcters are left untouched

def redact(message: str, terms: [str]) -> str:

''' In message, change each occurrence of a string in terms to a same-length string of Xs. '''

for t in message :

message = message.replace(t, redact\_term (t)) **x.replace(old, new)-> str**

**s.split (sep)** a list of substrings of strings s, obtained using delimiter string sep; with space

**s.strip()** removes leading/trailing spaces

**-sorted(mylist, key=cmp\_to\_key, reverse= True)— from greatest to least**

**TUPLE to LIST List to tuple**

>>> aTuple = ('A', 'B', 'C') # Creating a new tuple

>>> aList = ['1', '2', '3'] # Creating a new list

>>> aTupleAsAList = list(aTuple) # Casting a tuple to a list

>>> aListAsATuple = tuple(aList) # Casting a list to a tuple

Shoes = namedtuple('Shoes', 'brand size price')

Shoe1 = Shoes('Nikes', '10.5', 19.95)

## Shoe1.\_replace(price=29.95) THIS WILL NOT WORK, namedtuples are immutable

Shoe1 = Shoe1.\_replace(price=29.95) ## Now Shoe1 has a price of 29.95

Shoe2 = Shoe1.\_replace(price=29.95) ## Now Shoe2 has a price of 29.95 (use this method if you want to keep the Shoe1 namedtuple in tact)

**convert a list of strings into a list of namedtuples**

L = ["Taillevent/French/334-4433/Blanc de volaille/45.50", "Tour d'Argent/French/443-4444/Tournedos Rossini/55.00"]

and you'd want to produce

RL = [Restaurant("Taillevent","French","334-4433","Blanc de volaille", 45.50),

Here's the skeleton:

RL = [ ]

for s in L:

RL.append(string\_to\_Restaurant(s))

Restaurant("Tour d'Argent","French","443-4444","Tournedos Rossini", 55.00)