CSC1001 Final Cheat Sheet

Computer Architecture
CPU: CU – fetch commands ALU – execute commands
Memory: ROM, RAM
Number Converting
b-10 to b-2: divide – 2 | multiply – 2
Ex: 257.25 = 100000001.01_2
b-2 to b-8:

2	8	2	8	2	8	2	8
000	0	100	4	010	2	110	6
001	1	101	5	011	3	111	7

b-2 to b-16

2	16	2	16	2	16	2	16
0000	0	1000	8	0100	4	1100	C
0001	1	1001	9	0101	5	1101	D
0010	2	1010	A	0110	6	1110	E
0011	3	1011	B	0111	7	1111	F

Interactive: one line at a time
Script: in a file .py
Operator
Precedence: **()** - **func()** - **x[a:b]** - **x[a]** - ****** - **+a, -a** - ***, /, //, %** - **+, -** - **<< >>** - **bitwise &, ^, |** - **>= <= < >** - **== !=** - **assignment** – **identity** – **and, or, not** - LefttoRight
Floor div: 9 // 2.5 = 3
Divmod : divmod(257, 60) = 4, 17
** = right to left
Division return float,
Floor div // and mod return int,
Expression
Operators and operands to *produce some value* e.g. x = 1 + 2
Variables : storing values
start w _/abc, includes A-z, 0-9, _, case sens.
Reserved words:

```
False    None     True     and      as       assert   break
class    continue def      del      elif     else     except
finally  for      from     global   if       import   in
is       lambda   nonlocal not      or       pass     raise
return   try      while    with     yield
```

Cascaded x = y = z = 2
Simultaneous a, b = 8, 3
Augmented x += 2
Name error if variable is not found

Statements
A command, doesn't return any value, it directs the computer to do something
e.g. x = 1 + 3
Data Types
Converting type:
Value error if the value is not valid
int(s), float(s) convert to int/float (num!)
str(n) convert to string
eval(s) convert string to object/e xecuted
type(s) // return type of object
int("22") = 22, int(22.5) = 22,
int("22.2") = ValueError
Input Output
input("<prompt>: ")
print("%8.1f" %(variable), end="", sep="")
// 8.1 = 1 float point, use –8 to align left
default print: end="\n"

String: ordered group of chars, immutable
Defined using "" or ''
Concatenate: "a " + "b" = "a b"
Repetition (str*int): "a" * 4 = "aaaa"
Special strings: newlines \n, tab \t, \". \'
len(s) //r: length of string
s.center(i) // return centered list with length i
s.lower(), s.upper() // convert to u/l case
s.capitalize() // capitalize first letter only
s.title() // capitalize each word
s.count() // count occur of substr
s.replace('a', 'b', count) // r: replace all subst. a to b
s.find('a') // r: 1st occur. of subst., -1 not found, rfind() for last occur.
s.split('a') //r: <list> string split by 'a'
s.rsplit('a', maxsplit) // r: split by 'a' from right <maxsplit> times
s.strip() // r: strip all whitespaces, lstrip() = left, rstrip() = right
s.startswith('a') // r: <bool> if start with a or not
s.endswith('a') // r: <bool> if end with a or not
s.isalnum(), s.isalpha(), s.isdigit(), s.isdecimal()
s.join(<iterable>) // join all element in iterable with separator s, for dict, it joins the keys (must be str)
s[a:b] // slicing from a to b - 1
s[::-1] //reverse string
for s in string: // looping through string
Algorithms and Flowchart
Break: stop the current iteration, exit the loop
Continue: stop through current loop iteration
Flowchart
Sequential: top-down
Selection/Conditional: if else
Repetition: loop
Boolean: True/False
Operators: and (both true = True), or (one or both true = True)
bool(x) / return False for None, False, 0, empty list/tuple/string/dictionary
File
FileNotFoundError when file not found
file = open("file.txt", "<method>")
// method: r = read, w = write (from beg.), a = append (continue writing), x = create/return error if file exists
// rb/rt = read binary/ read text
f.read(size= (-1 def)) // r: specified data
f.readlines() // r: list of lines <list>
f.readline(size=s) // r: get first s char of line
f.write(s) // write s at current position
f.writelines(list) // write each element of the list as a line in the file
file.close() // close file
Comments / Syntaxing
Single line
''' ..... ''' multiline
Breakline using \
Errors
Syntax error (incorrect syntax, ***Type error*** (incorrect data type), ***Value error*** (correct type, incorrect value), ***Name error*** (no variable), ***Runtime error, Zerodivision error***
Exception handing (not for logic error)

```
try:
    // try this first
Except <Exc> :
    // when error occurs continue w this
else:
    // if not error, continue w this
finally:
   // will always be executed
```

Sequence
range(start, stop, step) // stop is not incl.
ex: range(1, 9, 2) = [1,3,5,7]
Operation:
Indexing: s[0] 0,1,2,…
Negative index: ending index
Slicing: s[start:stop:step]
s[2:] // from 2 to end
s[:5] // from beginning to no incl 5
s[::-1] // reverse
Membership: *in* or *not in*
Min/Max
List: mutable/can be modified, seq
Declared using []
[] = empty list
Accessing: l[index]
Index error if not found
Methods:
l.append(<el>) // add new element at the end
l.extend(<list>) // add list to the end
l.count(<el>) // r: number of elements <el>
l.index(<el>, start, end) //r: index of element <el>
l.insert(<i>, <el>) // insert an element at index <i>
l.pop(<i>) // remove at index <i>, return the removed element
l.remove(<el>) // remove first matching element
l.clear() // remove all elements from the list
del l[i] // delete sliced list
l.reverse() // reverse the list (no return)
l.sort(key=, reverse=) // sort the list (no return), reverse for descending order
sorted(l) // return sorted list
Tuples: immutable, seq.
Dict
Declared using {}
{} = empty dict
{key0: val0, key1: val1}
Accessing: dict["key"]
Keyerror if not found
Iterates: iterates the keys
for keys in dict:
d.keys() // return all keys of the dict.
d.values() // return all values of the dict
d.items() // return al (key, value) of the dict
d.pop(key, d) // remove item with key = key return the value
d.popitem() // remove last item in the dict returns (key, val)
d.get(key, d) // return the value of a key, return d (default if not found),
Memory Allocation / Variable Reference
Passing by ref (e.g., b = a) = same reference
list[:] alloc new memory, assignment alloc new memory (in function become local var)
String: same value same ref.
List: different var. assignment, dif. Ref

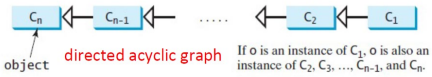
OOP
Variable in py = reference to obj with **id**
Methods: function for object, specific object
Object: entity with id, state (attributes [var]), behavior (methods)
Class: contract, template/blueprint of object
Obj. is **instance** of class, **instantiation** creating an instance
Method initializer **__init__()** to initialize obj. state
Use **constructor** to create object (1) create obj. in memory, (2) invoke initializer. Constructor **argument** = param of init without self
e.g. object = ClassName(argument)
All method in class use param **self**, refers to the object
Instance variable: data fields, specific var.
Instance methods: method for specific obj.
e.g. object.method(), object.variable
Private data fields: use self._var or self.__var, but can be called using a._class__var
Abstraction: separate the implementation from its usage.
Class abstraction: separating class implementation,
encapsulation: wrap all methods in a single entity, the process inside is hidden from user
Inheritance: defining new class from old class, passing the var, method to new class

- Subclass inherit from superclass. Every class in python inherit **object class** (by default) e.g., class Circle(GeometricObj):
- In the initializer need to initialize the super class e.g. super().__init__() or ClassName.__init__(self) **all super class also need super().__init__()*
- Subclass **can override** superclass method

__new__() auto invoked, then invoke __init__()
Default new in **object** class will (1) auto invoke new of superclass and if *cls* is passed (2) init of the subclass

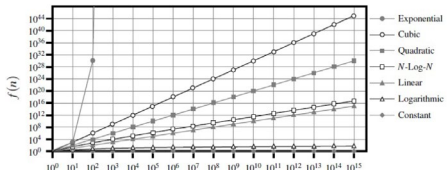
CSC1001 Final Cheat Sheet

__str__() return string desc (for print)
__eq__(), __add__(), __sub__(), __mul__()
Polymorpism: object of a subclass can be passed to a param of superclass type (same method name for different class, eg. Super and subclass)
Dynamic binding: method in several classes along inheritance chain is dynamically bound (priority chain: C1 subclass to Cn superclass)



Method Resolution Order: for multiple inheritance e.g. C(B,A) : C -> B -> A, D(C) : D -> C -> B -> A
isinstance(object, ClassName) to check whether is instance of class or not.
Mutability: pass object to a function and do modification. For mutable (list) the object will change, immutable (string, number, tuple) no.

Algorithm
Good algo: running time and space usage, depends on the size of the inputs
Challenges: different performance, limited test inputs, algo must be fully implemented
Principles:
(1) Counting primitive operation (constant rt) e.g., assignment, arithmetic operation, logic operation, indexing, caling and return function
(2) Measure operations as f(input size)
(3) Focusing on worst-case



Asymtotic analysis: focus on the growth rate as n bigger (disregard hardware performance)
We say that f(n) is O(g(n)) if there is a real constant c > 0 and an integer constant $n_0 \geq 1$ such that
$f(n) \leq cg(n)$, for $n \geq n_0$
Ignore **constant factors** and **lower order terms**.
e.g. 2^n+2 = O(2^n), 2n^2 + n = O(n^2)
Comparative analysis

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Tractability: differentiated by polynomial and exponential time
Recursive: need (1) base case (2) recursive case, call again the function itself.
In python, recursive stored in stack of record stores
1) Linear recursion: call recursion 1 time
2) Multiple recursion: call recursion > 1
Search Algorithm: Linear Search O(n), Binary Search O(log n) → n/2^r >= 1, r <= log2(n)
Data Structure
Stack: LIFO, can only access/remove the newest el.

```
class ListStack:

    def __init__(self):
        self.__data = list()

    def __len__(self):
        return len(self.__data)

    def is_empty(self):
        return len(self.__data) == 0
```

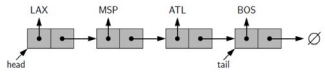
```
def push(self,e):
    self.__data.append(e)

def top(self):
    if self.is_empty():
        print('The stack is empty.')
    else:
        return self.__data[self.__len__()-1]

def pop(self):
    if self.is_empty():
        print('The stack is empty.')
    else:
        return self.__data.pop()
```

Queue: FIFO, can only access/remove first element in queue

```
class ListQueue:
    default_capacity = 5

    def __init__(self):
        self.__data = [None]*ListQueue.default_capacity
        self.__size = 0
        self.__front = 0
        self.__end = 0

    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size ==0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.__data[self.__front]
```

```
def dequeue(self):

    if self.is_empty():
        print('Queue is empty.')
        return None

    answer = self.__data[self.__front]
    self.__data[self.__front] = None
    self.__front = (self.__front+1) \
                   % ListQueue.default_capacity
    self.__size -=1
    return answer

def enqueue(self,e):
    if self.__size == ListQueue.default_capacity:
        print('The queue is full.')
        return None

    self.__data[self.__end] = e
    self.__end = (self.__end+1) \
                 % ListQueue.default_capacity
    self.__size += 1

def outputQ(self):
    print(self.__data)
```

Linked List
List in python referential, contain **ref only**
In other lang: compact array, storing bits
Array: store same data types
Singly Linked List



Traversing/link hopping/pointer hopping
Inserting: to head/tail, delete: only from head

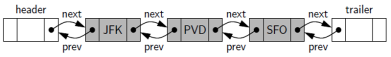
```
class Node:
    def __init__(self, element, pointer):
        self.element = element
        self.pointer = pointer

class LinkedStack:

    def __init__(self):
        self.head = None
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def push(self,e):
        self.head = Node(e, self.head)
        self.size += 1
```

```
def top(self):
    if self.is_empty():
        print('Stack is empty.')
    else:
        return self.head.element

def pop(self):
    if self.is_empty():
        print('Stack is empty.')
    else:
        answer = self.head.element
        self.head = self.head.pointer
        self.size -=1
        return answer
```

```
class LinkedQueue:

    def __init__(self):
        self. head = None
        self. tail = None
        self. size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.head.element
```

```
def dequeue(self):
    if self.is_empty():
        print('Queue is empty.')
    else:
        answer = self.head.element
        self.head = self.head.pointer
        self.size -= 1
        if self.is_empty():
            self.tail = None
        return answer

def enqueue(self,e):
    newest = Node(e,None)

    if self.is_empty():
        self.head = newest
    else:
        self.tail.pointer = newest
    self.tail = newest
    self.size += 1
```

Doubly Linked List

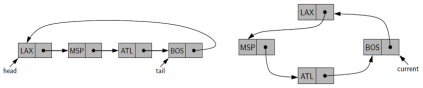


Allow to delete from the tail, add special nodes: header node and trailer node (sentinels)

```
def insert_between(self,e,predecessor,successor):
    newest = Node(e,predecessor,successor)
    predecessor.nxt = newest
    successor.prev = newest
    self.size+=1
    return newest

def delete_node(self,node):
    predecessor = node.prev
    successor = node.nxt
    predecessor.nxt = successor
    successor.prev = predecessor
    self.size -=1
    element = node.element
    node.prev = node.nxt = node.element = None
    return element

def iterate(self):
    pointer = self.header.nxt
    print('The elements in the list:')
    while pointer != self.trailer:
        print(pointer.element)
        pointer = pointer.nxt
```

```
def main():
    d=DLList()
    d.__len__()

    newNode = d.insert_between(10,d.header,d.t
    newNode = d.insert_between(20,newNode,d.tr
    newNode = d.insert_between(30,newNode,d.tr
    d.iterate()
    d.delete_node(d.header.nxt.nxt)
    d.iterate()

class Node:
    def __init__(self,element,prev,nxt):
        self.element = element
        self.prev = prev
        self.nxt = nxt

class DLList:

    def __init__(self):
        self.header = Node(None,None,None)
        self.trailer = Node(None,None,None)
        self.header.nxt = self.trailer
        self.trailer.prev = self.header
        self.size = 0
```

Circulary



Ex: Round-robin scheduler to allocate slices of CPU time to various applications
Sorting Algorithm
->**Bubble Sort**: iterate list compare i and i + 1, swap if larger, iterate again with ignoring last element [O(n^2)]
->**Quick Sort**: pick a pivot, partition around the pivot, eg 5 is pivot 4 2 3 |5| 8 7 6, then recursive sort the left and right. [O(n^2)], avg: O(nlogn)

```
def bubble(bubbleList):
    listLength = len(bubbleList)
    while listLength > 0:
        for i in range(listLength - 1):
            if bubbleList[i] > bubbleList[i+1]:
                buf = bubbleList[i]
                bubbleList[i] = bubbleList[i+1]
                bubbleList[i+1] = buf
        listLength -= 1
    return bubbleList


def main():
    bubbleList = [3, 4, 1, 2, 5, 8, 0, 100, 17]
    print(bubble(bubbleList))
```

```
def quickSort(L, low, high):
    i = low
    j = high
    if i >= j:
        return L
    key = L[i]
    while i < j:
        while i < j and L[j] >= key:
            j = j-1
        L[i] = L[j]
        while i < j and L[i] <= key:
            i = i+1
        L[j] = L[i]
    L[i] = key
    quickSort(L, low, i-1)
    quickSort(L, j+1, high)
    return L
```

Tree
Contain **root** (top element) with children node.
Root T has no **parent**, each node v besides the root has a parent node w (v child of w).
Edge: pair of nodes (u,v), **Path**: sequence of nodes (consecutive) that form an edge
Depth of v, the length of path from root to v
Leaf node: no child, **internal node**: has child
Binary tree: every node has at most 2 childr, left and right (left then right). Full binary trees: 0/2 child

```
class Node:

    def __init__(self,element, parent = None,\
                 left = None, right = None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right

class LBTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size
```

```
def find_root(self):
    return self.root

def parent(self,p):
    return p.parent

def left(self,p):
    return p.left

def right(self,p):
    return p.right

def num_child(self,p):
    count = 0
    if p.left is not None:
        count+=1
    if p.right is not None:
        count+=1
    return count
```

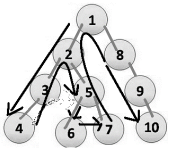
```
def add_root(self,e):
    if self.root is not None:
        print('Root already exists.')
        return None
    self.size = 1
    self.root = Node(e)
    return self.root

def add_left(self,p,e):
    if p.left is not None:
        print('Left child already exists.')
        return None
    self.size+=1
    p.left = Node(e,p)
    return p.left
```

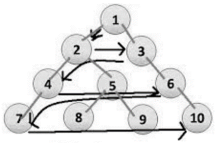
```
def add_right(self,p,e):
    if p.right is not None:
        print('Right child already exists.')
        return None
    self.size+=1
    p.right = Node(e,p)
    return p.right

def replace(self,p,e):
    old = p.element
    p.element = e
    return old

def delete(self,p):
    if p.parent.left is p:
        p.parent.left = None
    if p.parent.right is p:
        p.parent.right = None
    return p.element
```

DFS: tranversing as deep as possible to leaf node
BFS: starts from root, visit all the positions at depth d, then d + 1 until the max depth

```
def DFSearch(t):
    if t:
        print(t.element)
    if (t.left is None) and (t.right is None):
        return
    else:
        if t.left is not None:
            DFSearch(t.left)
        if t.right is not None:
            DFSearch(t.right)
```

```
def BFSearch(t):

    q = ListQueue()
    q.enqueue(t)

    while q.is_empty() is False:
        cNode = q.dequeue()
        if cNode.left is not None:
            q.enqueue(cNode.left)
        if cNode.right is not None:
            q.enqueue(cNode.right)
        print(cNode.element)
```





DFS : 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10
BFS: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10