PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

APRIL 2013 EXAMINATIONS

CSC 148 H1S
Instructor(s): F. Pitt

Duration — 3 hours

No Aids Allowed

You must achieve 40% on this final examination in order to pass the course.

A 2-sided Exam Aid Sheet is Attached.

Student Number: |_|_|_|_|_|_|_|_|_|_|

Last (Family) Name(s): ____________________

First (Given) Name(s): ____________________

Do **not** *turn this page until you have received the signal to start.*
In the meantime, please read the instructions below *carefully*.

This Final Examination paper consists of 8 questions on 22 pages (including this one), printed on both sides of the paper. *When you receive the signal to start, please make sure that your copy of the paper is complete and fill in your name and student number above.*

Answer each question directly on this paper, in the space provided, and use one of the "blank" pages (at the end of the paper) for rough work. If you need more space for one of your solutions, use a "blank" page and *indicate clearly the part of your work that should be marked.*

In your answers, you may use any of the Python built-in functions and standard modules listed on the accompanying "Python 3 Cheat Sheet." You must write your own code for everything else.

Comments and docstrings are *not required*, except where we ask for them explicitly. However, they may help us grade your answers and may be worth part marks if you cannot write a complete answer. Helper functions/methods are allowed, except where we explicitly forbid them.

If you are unable to answer a question (or part of a question), remember that you will get 10% of the marks for any solution that you leave *entirely blank* (or where you cross off everything you wrote to make it clear that it should not be marked).

MARKING GUIDE

1: ______/10

2: ______/10

3: ______/10

4: ______/10

5: ______/10

6: ______/10

7: ______/10

8: ______/10

BONUS
MARKS: ______/ 6

TOTAL: ______/80

Good Luck!

Question 1. [10 MARKS]

Consider the following description of a "real-world" system that you must represent in a program.

- A train has a sequence of cars, where each car is either a passenger car or a freight car.
- A passenger car can carry at most 75 passengers. The weight of a passenger car is the sum of the weights of the passengers in the car. Every passenger has the same, fixed weight: 75kg.
- A freight car can carry at most 6000kg. The weight of a freight car is the sum of the weights of its freight items. Every freight item has its own weight (an integer) in kg.
- We can add a car to a train (anywhere in the sequence of cars), remove a car from a train (anywhere in the sequence of cars), add and remove some number of passengers in a passenger car, add and remove individual freight items in a freight car, find out how many passengers are on the train, and find out the total weight of the train.

Write classes that model the system described above.

- For each class, write a short constructor (`__init__` method) with comments to explain each attribute.
- Write the method header (and a brief docstring) for every method, but **do *not* write code** for methods other than the constructors—just leave the method bodies empty.
- Your design must make appropriate use of inheritance.

To get you started, we wrote the constructor for class `Train` below.

```
class Train:
    def __init__(self):
        """Initialize this train to be empty (no cars)."""
        self.cars = []  # the list of cars in this train
```

Question 1. (CONTINUED)

Question 2. [10 MARKS]

Recall that a "Priority Queue" stores a sequence of items, each one associated with a *priority*. Suppose that you are given the following code in a file named priqueue.py.

```
class EmptyQueueError(Exception):
    pass

class PriQueue:
    """A Priority Queue."""

    def __init__(self):
        """Initialize this Priority Queue to be empty."""
        # ...code omitted...

    def is_empty(self):
        """Return True iff this Priority Queue is empty."""
        # ...code omitted...

    def insert(self, item, priority):
        """Insert item with priority (an int) into this Priority Queue."""
        # ...code omitted...

    def extract_min(self):
        """Remove and return the item with minimum priority in this Priority Queue;
        if more than one item has the same minimum priority, remove and return the
        one that was inserted first. Raise EmptyQueueError if this Priority Queue is
        empty."""
        # ...code omitted...
```

Write test cases for every method of the priqueue module. For each test case, state clearly: (1) the purpose of the test (what are you testing?), (2) a few lines of code to run the test case, and (3) the expected result (output or change in the contents). To get you started, we give an example below. For full marks, you only need 2 or 3 well-chosen test cases for each method.

```
# Example test: check that __init__ initializes the priority queue to be empty.
>>> q = PriQueue()
>>> q.is_empty()
True  # expected result
```

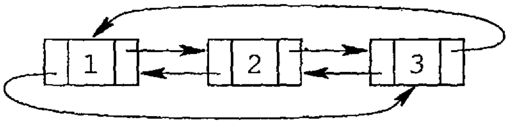
Question 2. (continued)

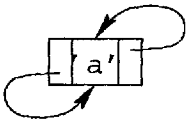
Question 3. [10 MARKS]

In a *doubly-linked* list, each node stores *three* references: one to the item stored in the node, one to the node immediately before it, and one to the node immediately after it. This can be coded as follows.

```
class DLNode:
    """A node in a doubly-linked list."""
    def __init__(self, item, prev=None, next=None):
        self.item = item  # the item in this node
        self.prev = prev  # the node before this one in the list (another DLNode)
        self.next = next  # the node after this one in the list (another DLNode)
```

In a *circular* doubly-linked list, the last node in the list has its next attribute set to refer back to the first node, and the first node in the list has its prev attribute set to refer forward to the last node. For example, the pictures below show a circular doubly-linked list that stores the items 1, 2, 3 (on the left) and a circular doubly-linked list that stores the single item 'a' (on the right).





Write the body of each method below (and on the next page) to satisfy their docstrings. Assume that class DLNode above is in the same file as class DLList below. Include comments to explain your design. (HINT: draw pictures!)

```
class DLList:
    """A circular doubly-linked list."""

    def __init__(self):
        """Initialize this list to be empty."""
```

```
    def __len__(self):
        """Return the number of items in this list."""
```

Question 3. (CONTINUED)

```
def insert(self, i, item):
    """Insert item at index i. Raise IndexError if i < 0 or i > len(self). For
    example, calling insert(1, 7) on the first list from the previous page changes
    the list to have values 1, 7, 2, 3."""
```

```
def remove(self, i):
    """Remove the item at index i. Raise IndexError if i < 0 or i > len(self) - 1.
    For example, calling remove(1) on the first list from the previous page changes
    the list to have values 1, 3."""
```

Question 4. [10 MARKS]

Part (a) [2 MARKS]

Write a short, precise definition of "Binary Search Tree". In your answer, you may use the term "binary tree" without explaining what it means.

Part (b) [4 MARKS]

Consider the following node class for binary trees.

```
class BTNode:
    """A node in a binary tree."""
    def __init__(self, item, left=None, right=None):
        self.item = item  # the item stored in this node
        self.left = left  # the left child of this node (another BTNode)
        self.right = right  # the right child of this node (another BTNode)
        self.parent = None  # place-holder for the parent of this node (another BTNode)
```

Complete the *method* below to satisfy its docstring.

```
    def set_parents(self):
        """Set the parent attribute of every node in the subtree rooted at this node,
        except for self (the parent of self is unchanged)."""
```

Question 4. (CONTINUED)

Part (c) [4 MARKS]

Consider the following node class for *general trees* (where each node can have a different number of children, with no fixed limit on the number of children).

```
class TreeNode:
    """A node in a general tree."""
    def __init__(self, item, children):
        self.item = item  # the item stored in this node
        self.children = children[:]  # a copy of the list of children, where each
                                     # element of children is a TreeNode
```

Complete the *function* below to satisfy its docstring.

```
def branching(root):
    """Return the branching factor of the general tree rooted at root. Recall that the
    branching factor is the maximum number of children of any node in the tree."""
```

Question 5. [10 MARKS]

Part (a) [2 MARKS]

Write a short, precise definition of "Min-Heap". In your answer, you may use the term "binary tree" without explaining what it means.

Part (b) [2 MARKS]

Draw the *tree structure* for the heap stored in the list below. (Of course, don't just draw a tree with "blank" nodes: include each of the values in your picture!)

5	20	7	32	40	9	8	35	

Question 5. (CONTINUED)

Part (c) [3 MARKS]

The first list below stores a heap. Perform the operation `insert(14)` on this heap and write down the resulting heap in the second list. (Use the space below the lists to draw pictures of the heap, to help you trace through the operation.)

initial:	5	20	7	32	40	9	8	35	
result:									

Part (d) [3 MARKS]

The first list below stores a heap. Perform the operation `extract_min()` on this heap and write down the resulting heap in the second list. (Use the space below the lists to draw pictures of the heap, to help you trace through the operation.)

initial:	5	20	7	32	40	9	8	35	
result:									

Question 6. [10 MARKS]

Part (a) [5 MARKS]

Draw a *detailed* picture of the memory model for the following code. Make sure to represent **every** value as a separate object and to indicate clearly the type and memory address of each object.

```
class BSTNode:
    def __init__(self, item, left=None, right=None):
        self.item, self.left, self.right = item, left, right

class BST:
    def __init__(self, container=[]):
        self.root = None
        for item in container:
            self.insert(item)
    # ...standard implementation for __contains__, insert, remove...

tree = BST([4, 2, 6])
# DRAW YOUR PICTURE AT THIS POINT IN THE EXECUTION OF THE CODE.
```

Question 6. (CONTINUED)

Part (b) [5 MARKS]

Draw a *detailed* picture of the memory model for the following code. Make sure to represent **every** value as a separate object and to indicate clearly the type and memory address of each object.

```
class Heap:
    def __init__(self, container=[]):
        self.items = container[:]
        self.size = len(container)
        self.heapify()
    # ...standard implementation for insert, get_min, extract_min, heapify...

heap = Heap([4, 2, 6])
# DRAW YOUR PICTURE AT THIS POINT IN THE EXECUTION OF THE CODE.
```

Question 7. [10 marks]

Consider the following implementation of binary search.

```
def bin_search(L, x):
    """Return True iff value x is in list L. Precondition: L is sorted."""

    while len(L) > 1:
        mid = len(L) // 2
        if x < L[mid]:
            L = L[:mid]
        else:
            L = L[mid:]

    return len(L) > 0 and L[0] == x
```

Part (a) [4 marks]

What is the worst-case running time of `bin_search`? Justify your answer. (Hint: remember to account for the slicing operator `[:]`!)

Question 7. (continued)

Part (b) [4 marks]

Write a function `fast_search` that performs binary search in the same way as `bin_search` (from the previous part), but more efficiently.

Part (c) [2 marks]

What is the worst-case running time of your function `fast_search` above? Justify your answer.

Question 8. [10 MARKS]

Consider the following class for nodes in a skip list (used in Part 2 of the project, though you might have chosen different names for the attributes).

```
class SkipNode:
    """A node in a skip list."""
    def __init__(self, item, skip, right, down):
        self.item = item  # the item stored in this node
        self.skip = skip  # the difference between the index of the next node on this
                          # level and the index of this node
        self.right = right  # the next node on this level
        self.down = down  # the node storing item one level below
```

Complete the function below to satisfy its docstring. Write your code to be as efficient as possible (making use of the skip list properties), and *include comments to explain what you are doing* (in particular, state clearly any assumption you make about the structure of the skip list and its nodes).

```
def index(head, item):
    """Return the index of item in the skiplist starting at node head (where the index
    of head is -1). Raise ValueError if item is not in the skip list."""
```

Question 8. (continued)

over...

Bonus. [6 MARKS]

WARNING! This question is difficult and will be marked harshly: credit will be given only for making *significant* progress toward a correct answer — in particular, you will receive *no credit* for leaving this blank. Please attempt this only *after* you have completed the rest of the Final Examination.

Write a function is_complete that takes the root of a binary tree and returns True iff that tree is complete — **without using recursion**! (You will receive *at most* 2 marks if your solution uses recursion.) Of course, you are allowed to use other data structures and helper functions. But none of your code can be recursive, and you must write the code for anything you need that is not built-in. To get you started, here is class BTNode (you may add code to this class, if you find it useful).

```
class BTNode:
    """A node in a binary tree."""
    def __init__(self, item, left=None, right=None):
        self.item = item  # the item stored in this node
        self.left = left  # the left child of this node (another BTNode)
        self.right = right  # the right child of this node (another BTNode)
```

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number.

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere. ***Clearly label each such answer with the appropriate question and part number.***

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number.

Please write nothing on this page.

Python 3 Cheat Sheet

Official Python documentation on
http://docs.python.org/py3k

Base Types

integer, float, boolean, string

```
  int   783    0     -192
float   9.23   0.0   -1.7e-6
                         10⁻⁶
 bool   True   False
  str   "One\nTwo"     'I\'m'
            new line    ' escaped
        multiline """X\tY\tZ
                  1\t2\t3"""
                     tab char
```

immutable, ordered sequence of chars

Container Types

- ordered sequence, fast index access, repeatable values

```
   list  [1,5,9]    ["x",11,8.9]   ["word"]    []
  tuple  (1,5,9)     11,"y",7.4    ("word",)   ()
```

immutable — expression with just comas

str as an ordered sequence of chars

- no *a priori* order, unique key, fast key access ; keys = base types or tuples

```
   dict  {"key":"value"}                            {}
         {1:"one",3:"three",2:"two",3.14:"π"}
    set  {"key1","key2"}     {1,9,3,0}          set()
```

dictionary — *key/value associations*

Identifiers

for variables, functions, modules, classes… names

a..zA..Z_ followed by **a..zA..Z_0..9**

- ▫ diacritics allowed but should be avoided
- ▫ language keywords forbidden
- ▫ lower/UPPER case discrimination

☺ **a toto x7 y_max BigOne**

☹ ~~8y~~ ~~and~~

Variables assignment

```
x = 1.2+8+sin(0)
```

value or computed expression — variable name (identifier)

```
y,z,r = 9.2,-7.6,"bad"
```

variables names — container with several values (here a tuple)

```
x+=3     increment
x-=2     decrement
x=None   « undefined » constant value
```

Conversions

type (*expression*)

```
int("15")       can specify integer number base in 2nd parameter
int(15.56)      truncate decimal part (round(15.56) for rounded integer)
float("-11.24e8")
str(78.3)       and for litteral representation ⟶ repr("Text")
```

see other side for string formating allowing finer control

```
bool ⟶ use comparators (with ==, !=, <, >, …), logical boolean result
list("abc") ⟶ ['a','b','c']                 use each element from sequence
dict([(3,"three"),(1,"one")]) ⟶ {1:'one',3:'three'}
set(["one","two"]) ⟶ {'one','two'}          use each element from sequence
":".join(['toto','12','pswd']) ⟶ 'toto:12:pswd'
```

joining string — sequence of strings

```
"words with spaces".split() ⟶ ['words','with','spaces']
"1,4,8,2".split(",") ⟶ ['1','4','8','2']
```

splitting string

Sequences indexing

for lists, tuples, strings, …

negative index	-6	-5	-4	-3	-2	-1	
positive index	0	1	2	3	4	5	
lst=	[11,	67,	"abc",	3.14,	42,	1968]	
positive slice	0	1	2	3	4	5	6
negative slice	-6	-5	-4	-3	-2	-1	

```
len(lst) ⟶ 6
```

individual access to items via [*index*]

```
lst[1]→67        lst[0]→11     first one
lst[-2]→42       lst[-1]→1968  last one
```

access to sub-sequences via [*start slice* : *end slice* : *step*]

```
lst[:-1]→[11,67,"abc",3.14,42]     lst[1:3]→[67,"abc"]
lst[1:-1]→[67,"abc",3.14,42]       lst[-3:-1]→[3.14,42]
lst[::2]→[11,"abc",42]             lst[:3]→[11,67,"abc"]
lst[:]→[11,67,"abc",3.14,42,1968]  lst[4:]→[42,1968]
```

Missing slice indication → from start / up to end.

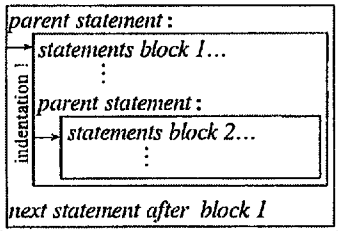
On mutable sequences, usable to remove del lst[3:5] *and to modify with assignment* lst[1:4]=['hop',9]

Boolean Logic

Comparators: < > <= >= == !=
≤ ≥ = ≠

```
a and b   logical and
          both simultaneously
a or b    logical or
          one or other or both
not a     logical not
True      true constant value
False     false constant value
```

Statements Blocks



parent statement :
statements block 1…
⋮
parent statement :
statements block 2…
⋮
next statement after block 1

Maths

floating point numbers… approximated values! *angles in radians*

Operators: + - * / // % **
× ÷ integer ÷ ÷ remainder a^b

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
```

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
acos(0.5)→1.0471…
sqrt(81)→9.0     √
log(e**2)→2.0
```

etc. (cf doc)

Conditional Statement

statements block executed only if a condition is true

if *logical expression* **:**
⟶ *statements block*

can go with several elif, elif... and only one final else, example :

```
if x==42:
    # block if logical expression x==42 is true
    print("real truth")
elif x>0:
    # else block if logical expression x>0 is true
    print("be positive")
elif bFinished:
    # else block if boolean variable bFinished is true
    print("how, finished")
else:
    # else block for other cases
    print("when it's not")
```

Conditional loop statement

statements block executed as long as condition is true

while *logical expression* **:**
→| *statements block*

```
s = 0 ⎫
i = 1 ⎭ initializations before the loop
```

condition with at least one variable value (here `i`*)*

```
while i <= 100:
    # statement executed as long as i ≤ 100
    s = s + i**2
    i = i + 1 } ☝ make condition variable change
print("sum:",s) } computed result after the loop
```

$$s=\sum_{i=1}^{i=100} i^2$$

☝ *be careful of inifinite loops !*

Loop control

`break` *immediate exit*

`continue` *next iteration*

Iterative loop statement

statements block executed for each item of a container or iterator

for *variable* **in** *sequence* **:**
→| *statements block*

Go over sequence's values

```
s = "Some text" ⎫ initializations before the loop
cnt = 0         ⎭
```

loop variable, value managed by `for` *statement*

```
for c in s:
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Count number of `e` *in the string*

loop on dict/set = loop on sequence of keys
use slices to go over a subset of the sequence

Go over sequence's **index**

□ modify item at index
□ access items around index (before/after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Limit values greater than 15, memorization of lost values.

Go simultaneously over sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

Display / Input

```
print("v=",3,"cm :",x,",",y+4)
```

items to display: litteral values, variables, expressions

`print` options:

- □ **`sep=" "`** (items separator, default space)
- □ **`end="\n"`** (end of print, default new line)
- □ **`file=f`** (print to file, default standard output)

```
s = input("Instructions:")
```

☝ `input` always returns a **string**, convert it to required type (cf boxed *Conversions* on on ther side).

Operations on containers

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → sorted *copy*
`val in c` → boolean, membersihp operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index,value)

*Note: For dictionaries and set, these operations use **keys**.*

*Special for **sequence containeurs** (lists, tuples, strings) :*

`reversed(c)` → reverse *iterator*   `c*5` → duplicate   `c+c2` → concatenate
`c.index(val)` → position   `c.count(val)` → events count

Operations on lists

☝ modify original list

`lst.append(item)`	add item at end
`lst.extend(seq)`	add sequence of items at end
`lst.insert(idx,val)`	insert item at index
`lst.remove(val)`	remove first item with value
`lst.pop(idx)`	remove item at index and return its value
`lst.sort()` `lst.reverse()`	sort / reverse list *in place*

Operations on dictionaries

`d[`*key*`]=`*value*   `d.clear()`
`d[`*key*`]`→*value*   `del d[`*clé*`]`
`d.update(d2)` { *update/add associations*
`d.keys()`
`d.values()` } *views on keys, values associations*
`d.items()`
`d.pop(`*clé*`)`

Operations on sets

Operators:

- `|` → union (vertical bar char)
- `&` → intersection
- `-` `^` → difference/symetric diff
- `<` `<=` `>` `>=` → inclusion relations

`s.update(s2)`
`s.add(`*key*`)`   `s.remove(`*key*`)`
`s.discard(`*key*`)`

Files

storing data on disk, and reading it back

```
f = open("fil.txt","w",encoding="utf8")
```

file variable for operations

name of file on disk (+path…)

opening **mode**
- □ `'r'` read
- □ `'w'` write
- □ `'a'` append…

encoding of chars for text files:
`uft8` `ascii`
`latin1` …

cf functions in modules `os` and `os.path`

writing

```
f.write("hello")
```

☝ *text file → read /write only strings, convert from/to required type.*

reading

```
s = f.read(4)
```

empty string if end of file; if char count not specified, read whole file

read next line

```
s = f.readline()
```

```
f.close()
```

☝ don't forget to close file after use

Pythonic automatic close : `with open(…) as f:`

very common: iterative loop reading lines of a text file

```
for line in f :
    # line processing block
```

Generator of int sequences

frequently used in `for` *iterative loops*

default 0 — not included

`range` *([start,]stop [,step])*

```
range(5)       ────→ 0 1 2 3 4
range(3,8)     ────→ 3 4 5 6 7
range(2,12,3)  ────→  2 5 8 11
```

`range` returns a « generator », converts it to list to see the values, example:
`print(list(range(4)))`

Function definition

function name (identifier)

named parameters

```
def fctname(p_x,p_y,p_z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`return res` ← result value of the call. if no computed result to return: `return None`

☝ parameters and all of this bloc only exist *in* the block and *during* the function call (*"black box"*)

Function call

```
r = fctname(3,i+2,2*i)
```

one argument per parameter

retrieve returned result (if necessary)

Strings formating

formating directives — values to format

`"model {} {} {}".format(x,y,r)` ——→ `str`

`"{`*selection* `:` *formating* `!` *conversion*`}"`

□ **Selection :**

```
2
x
0.nom
4[key]
0[2]
```

Examples

```
"{:+2.3f}".format(45.7273)
→'+45.727'
"{1:>10s}".format(8,"toto")
→'      toto'
"{!r}".format("I'm")
→'"I\'m"'
```

□ **Formating :**

fillchar *alignment* *sign* *minwidth* . *precision~maxwidth* *type*

`< > ^ =`   `+ -` *space*   0 at start for filling with 0

integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…
float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default), `%` percent
string : `s` …

□ **Conversion** : `s` (readable text) or `r` (litteral representation)