Question 1. True or False

Circle **T** if the statement is true, otherwise circle **F** if the statement is false.

1. Method resolution order helps Python deal with the repeated base class problem. **T** **F**

2. All Python built-in methods go through type slots. **T** **F**

3. A programming language without multiple inheritance cannot implement mixins. **T** **F**

4. Python does not have class scope. **T** **F**

5. You can mix the use of fully qualified names and super() in cooperative inheritance. **T** **F**

Question 2. Multiple Choices

Pick all answer(s) that are correct.

a) Which of the following are true about Python mixins?

i. It requires subclass to complete its implementation.

ii. It can contain both member variables and functions.

iii. It is used as a super type to the derived class.

iv. Using it requires method forwarding.

v. The order in which mixins are inherited may change behaviour of the subclass.

b) Which of the following statements about method resolution order (MRO) are true?

i. Object-oriented programming languages that perform late binding must implement MRO.

ii. Depth-first search, left to right, respects local precedence order.

iii. In Python, method resolution order is also used on data attributes.

iv. C3 linearization cannot fail while creating method resolution order.

v. Linearization is the result of applying method resolution order to inheritance hierarchy.

Question 3. Short Answer

a) List two properties that C3 Linearization guarantees. You must explain what each guarantee means to receive full marks (e.g. with an example).

Preserves local precedence order – i.e. the order in which the parent classes are inherited

Preserves monotonicity – linearization of a class will not change order regardless of the inheritance hierarchy it may be in

b) For the following inheritance hierarchy in Python, draw a diagram of the hierarchy. Then compute, by hand, the C3 Linearization of class X.

```
class A: pass
class B: pass
class C: pass
class D: pass
class E: pass
class P(A, B, C): pass
class Q(D, B, E): pass
class R(D, A): pass
class X(P, R, Q): pass
```

```
L[A] = (A, o)
L[B] = (B, o)
L[C] = (C, o)
L[D] = (D, o)
L[E] = (E, o)
L[P] = (P, A, B, C, o)
L[Q] = (Q, D, B, E, o)
L[R] = (R, D, A, o)
L[X]
= (X, merge((P, A, B, C, o), (R, D, A, o), (Q, D, B, E, o), (P, R, Q)))
= (X, P, merge((A, B, C, o), (R, D, A, o), (Q, D, B, E, o), (R, Q)))
= (X, P, R, merge((A, B, C, o), (D, A, o), (Q, D, B, E, o), (Q)))
# A bad head, in tail of 2nd list
= (X, P, R, Q, merge((A, B, C, o), (D, A, o), (D, B, E, o)))
= (X, P, R, Q, D, merge((A, B, C, o), (A, o), (B, E, o)))
= (X, P, R, Q, D, A, merge((B, C, o), (o), (B, E, o)))
= (X, P, R, Q, D, A, B, merge((C, o), (o), (E, o)))
= (X, P, R, Q, D, A, B, C, E, o)
```

c) Given:

```
class A : foo = 1
class B : qux = 2
class W(A) : baz = 3
class X(A) : bar = 4
class Y(B) : foo = 5
class Z(B) : baz = 6
```

```
class N(type) : foo = 11
class M(N) : qux = 7
class P(W, X, Y) : pass
class Q(X, Y, Z): bar = 9
class R(W, Z) : bar = 0
class H(P, Q, R, metaclass=M) : pass
```

i. What is the C3 Linearization of H?

```
L[A] = (A, o)
L[B] = (B, o)
L[W] = (W, A, o)
L[X] = (X, A, o)
L[Y] = (Y, B, o)
L[Z] = (Z, B, o)
L[P] = (P, merge((W, A, o), (X, A, o), (Y, B, o), (W, X, Y)))
L[P] = (P, W, X, merge((A, o), (A, o), (Y, B, o), (Y)))
L[P] = (P, W, X, A, merge((o), (o), (Y, B, o), (Y)))
L[P] = (P, W, X, A, Y, B, o)
L[Q] = (Q, merge((X, A, o), (Y, B, o), (Z, B, o), (X, Y, Z)))
L[Q] = (Q, X, A, merge((o), (Y, B, o), (Z, B, o), (Y, Z)))
L[Q] = (Q, X, A, Y, Z, B, o)
L[R] = (R, merge((W, A, o), (Z, B, o), (W, Z)))
L[R] = (R, W, A, Z, B, o)
L[H] = (H, merge((P, W, X, A, Y, B, o), (Q, X, A, Y, Z, B, o),
          (R, W, A, Z, B, o), (P, Q, R)))
L[H] = (H, P, Q, R, merge((W, X, A, Y, B, o), (X, A, Y, Z, B, o),
          (W, A, Z, B, o)))
L[H] = (H, P, Q, R, W, X, merge((A, Y, B, o), (A, Y, Z, B, o),
          (A, Z, B, o)))
L[H] = (H, P, Q, R, W, X, A, Y, merge((B, o), (Z, B, o),
          (Z, B, o)))

L[H] = (H, P, Q, R, W, X, A, Y, Z, B, o)
```

ii. What are the values of H.foo, H.bar, H.baz, and H.qux?

foo: **1** bar: **9**

baz: **3** qux: **2**

Question 4. Programming Questions

a) Base on the following example, write a SocketLogMixin that will send log messages to a server, and a FileLogMixin that will save log messages in a local file.

```
import math

# abstract base class
class LogMixin:
    def __init__(self, log_level, **kwargs):
        self._level = log_level
        # do not pass kwargs to object base class

    def log(self, level, msg):
        if self._level < level:
            self._write(msg)    # child class must implement this

class ConsoleLogMixin(LogMixin):
    def _write(self, msg):
        print(msg)

class Employee:
    def __init__(self, name, **kwargs):
        self.name = name
        super().__init__(**kwargs)

class Accountant(Employee, ConsoleLogMixin):
    def __init__(self, name, **kwargs):
        kwargs['name'] = name
        kwargs['log_level'] = 2
        super().__init__(**kwargs)

    def make_payment(self, amount):
        level = math.log10(amount)
        self.log(level, "%s paid $%.2f"%(self.name, amount))
```

```
# example usage
ann = Accountant("Ann", log_file="transaction.txt",
                        log_server="127.0.0.1:8888")
ann.make_payment(420.69)
ann.make_payment(49.99)
ann.make_payment(1000.01)
```

```
class FileLogMixin(LogMixin):
    def __init__(self, log_file, **kwargs):
        self.file = open(log_file, "wt")
        super().__init__(**kwargs)

    def _write(self, msg):
        self.file.write(msg + '\n')

    def __del__(self):
        self.file.close()

import socket

class SocketLogMixin(LogMixin):
    def __init__(self, log_server, **kwargs):
        host, port = log_server.split(":")
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect((host, int(port)))
        super().__init__(**kwargs)

    def _write(self, msg):
        self.sock.send(msg + '\n')

    def __del__(self):
        self.sock.close()
```

b) Create a mixin named Indexable that will automatically forward `__setitem__` and `__getitem__` to an instance variable named data.

```
class Indexable:
    def __setitem__(self, index, value):
        self.data[index] = value

    def __getitem__(self, index):
        return self.data[index]
```

c) Suppose there exists two classes, Calendar and Clock, with the following interface (assume they have been implemented):

```
class Clock:
  def __init__(self, h, m, s)
  def __str__(self)

  # moves time forward by 1 second
  # returns True if clock resets,
  # i.e. from 23:59:59 to 00:00:00,
  # False otherwise.
  def tick()
```

```
class Calendar:
  def __init__(self, y, m, d)
  def __str__(self)

  # moves time forward by 1 day
  def advance()
```

Implement a class, CalendarClock, that is derived from Calendar and Clock shown above, and implement three methods, `__init__`, `__str__`, and `tick`. You must reuse existing functionality as much as possible, without making assumptions about how each super class stores their data attributes.

```
class CalendarClock(Calendar, Clock):
    def __init__(self, year, month, day, hour, minute, second):
        Calendar.__init__(self, year, month, day)
        Clock.__init__(self, hour, minute, second)

    def __str__(self):
        return Calendar.__str__(self) + ", " + Clock.__str__(self)

    def tick(self):
        if super().tick():
            super().advance()
```