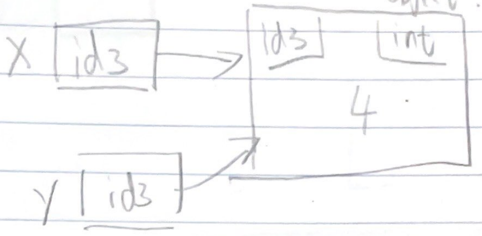
1/9 Lec 1

Assignment operator.

x=4 y=X point to some object.



- '=' : reassign.

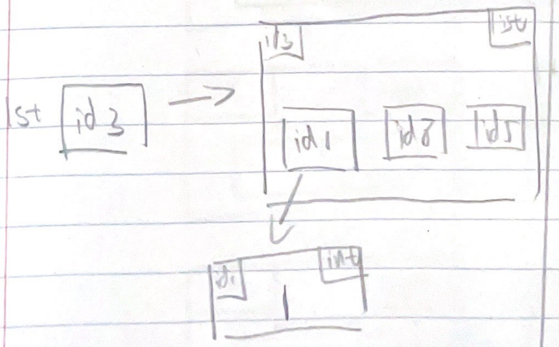
x=4
y=x
x=2

- reassign is not value changing, it is assign to new obj

immutable types: int, float .....

- Reassign is value changing.

lst= [1, 6, 3]
lst [0] = 8



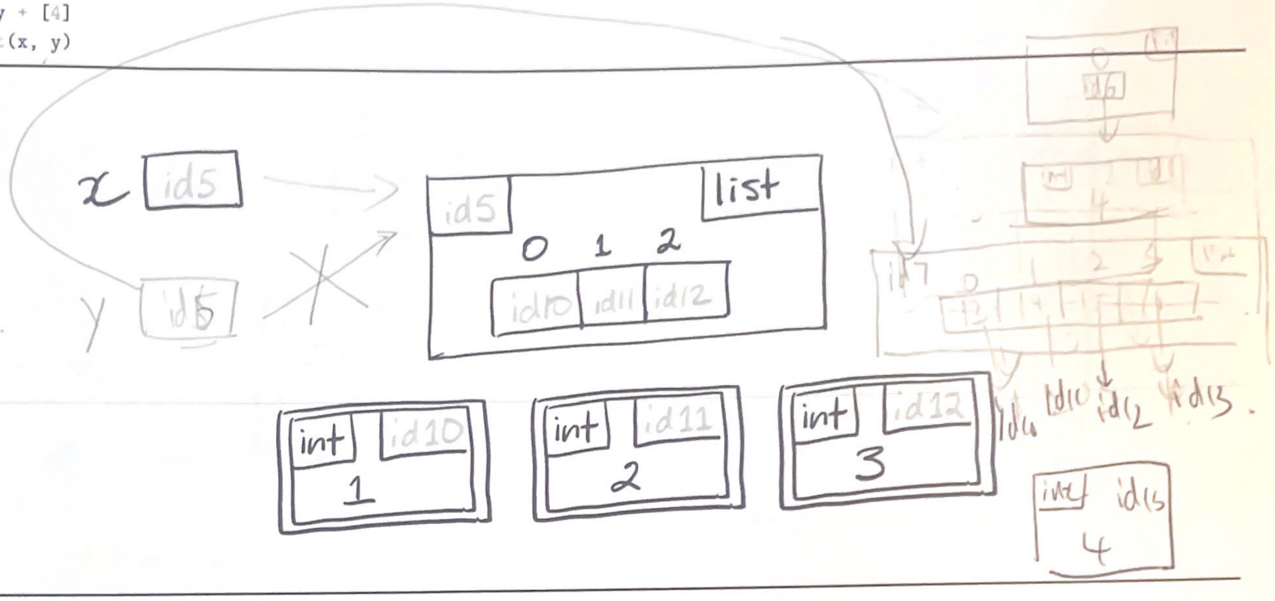
- Call stack

Mutable types: List, Set, Dictionary, Custom classes lec1

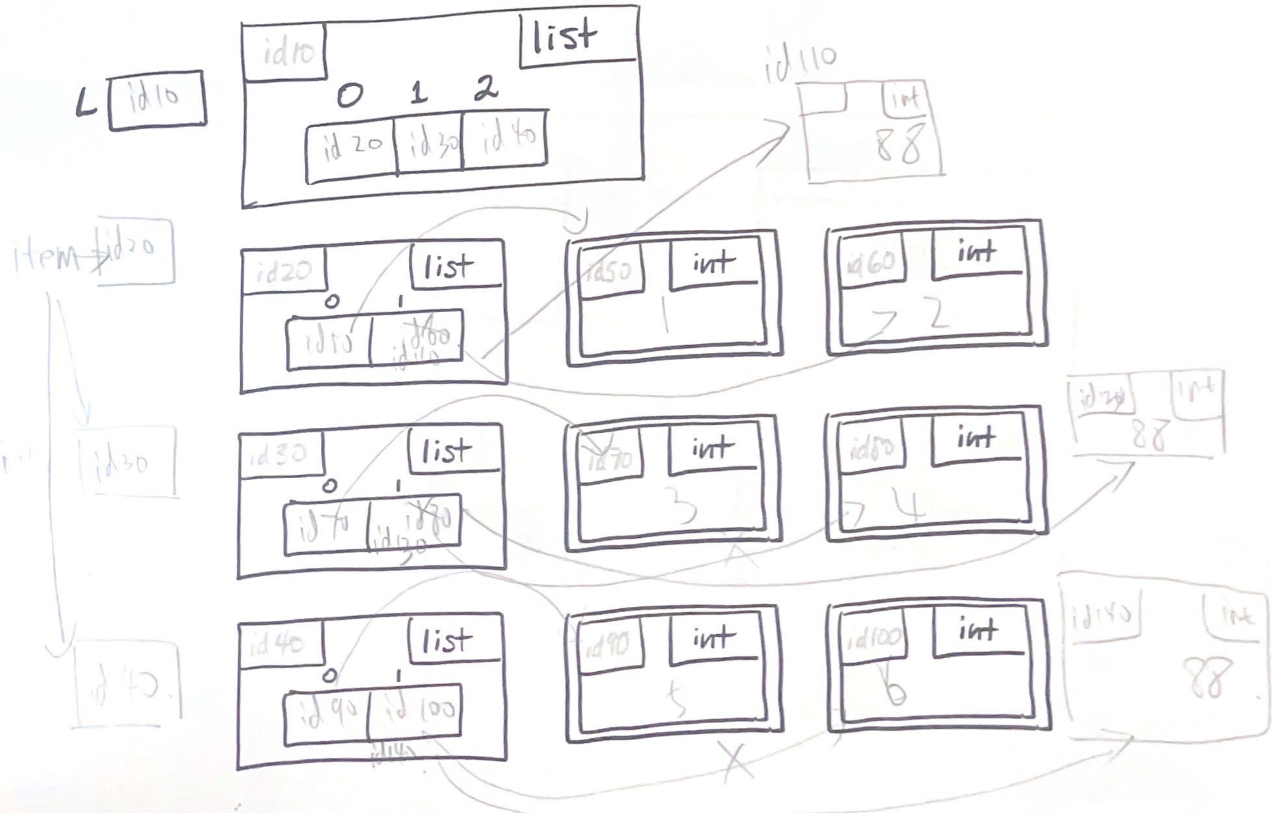
CSC148 - Python recap: Tracing simple code

For each snippet of code below, complete the memory model diagram and write what the output would be.

```
# Q1.
x = [1, 2, 3]
y = x
y = y + [4]
print(x, y)
```

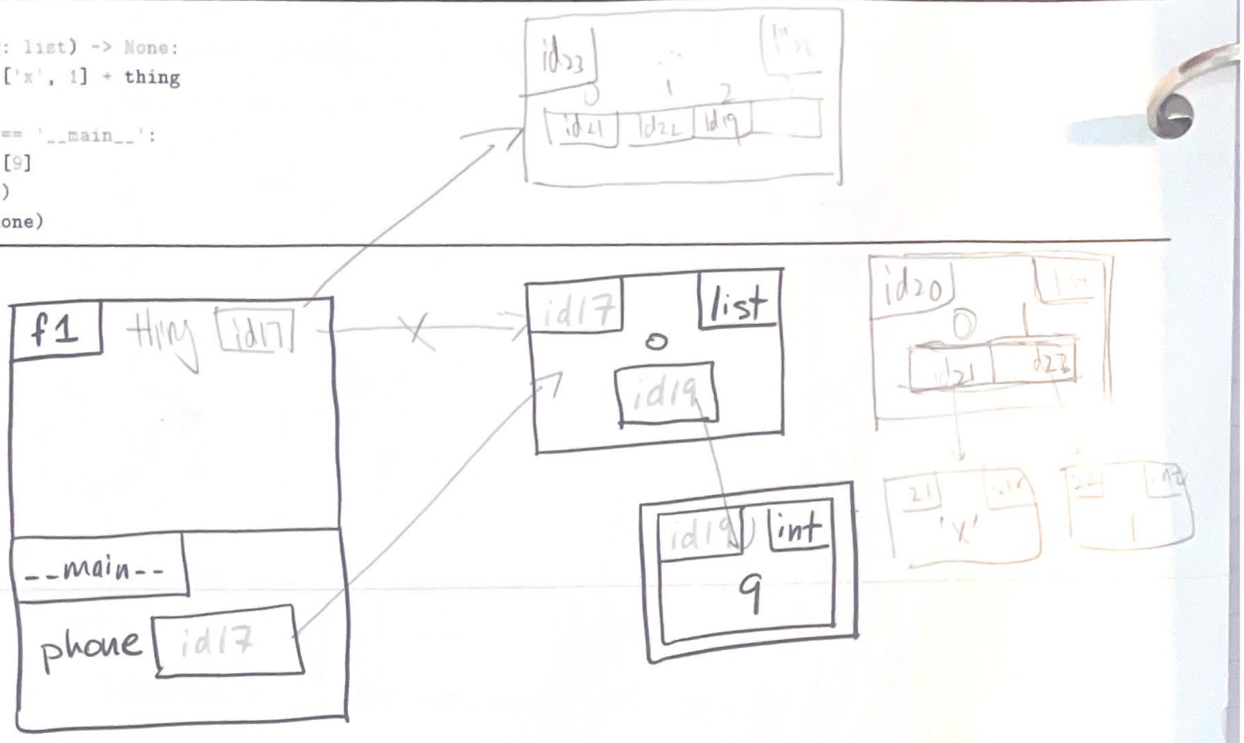


```
# Q2.
L = [[1, 2], [3, 4], [5, 6]]
for item in L:
    item[1] = 88
print(L)
```



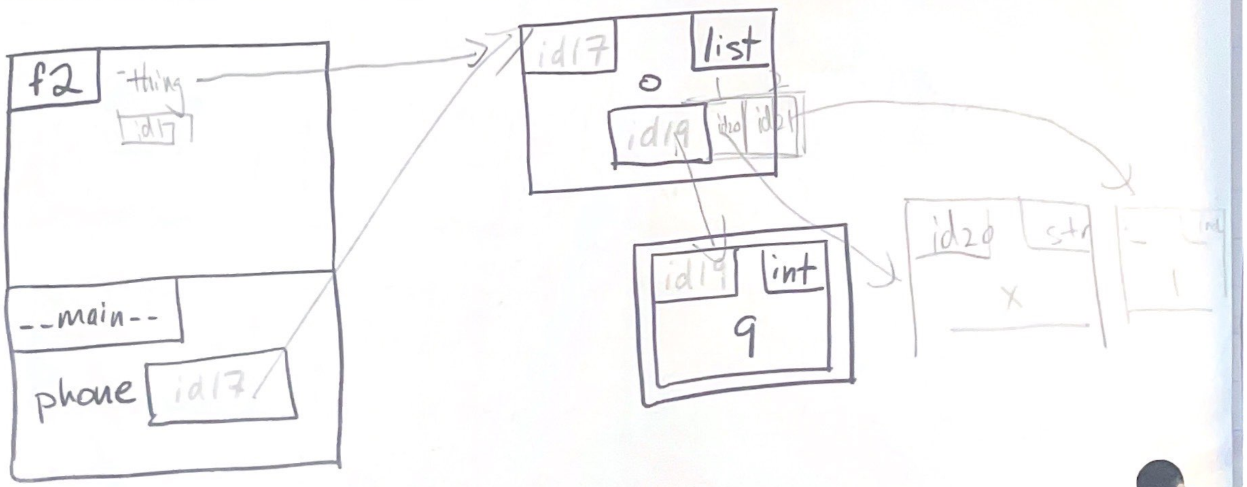
```
# Q3.
def f1(thing: list) -> None:
    thing = ['x', 1] + thing

if __name__ == '__main__':
    phone = [9]
    f1(phone)
    print(phone)
```



```
# Q4.
def f2(thing: list) -> None:
    thing.extend(['x', 1])

if __name__ == '__main__':
    phone = [9]
    f2(phone)
    print(phone)
```



Lec1

CSC148 - Choosing Test Cases

```
def insert_after(lst: List[int], n1: int, n2: int) -> None:
    """After each occurrence of <n1> in <lst>, insert <n2>.

    >>> lst = [5, 1, 2, 1, 6]
    >>> insert_after(lst, 1, 99)
    >>> lst
    [5, 1, 99, 2, 1, 99, 6]
    """
```

1. We can only test a tiny fraction of all possible calls to this method. Some properties are likely not relevant, such as whether the values in `lst` are positive or negative. In the table below, we have named one property that *is* relevant. Add more.

Relevant Property	Values to Try
position of `n1` in `lst`	front, back, somewhere else
number of occurance of n1	0, 1, 2 ., all
is there adjacent n1 ?	Yes , no .
n1 = n2 ?	Yes , no .

2. Using the above table, now define some specific test cases. Again, we have started. Add more. Continue on the reverse if needed.

lst	n1	n2	Purpose
`[0, 1, 2, 3]`	0	99	`n1` at the front
`[0, 1, 2, 3]`	3	99	`n1` at the back
`[0, 1, 2, 3]`	1	99	`n1` somewhere else
[0 , 2,3] [0,1,2,1,3,1] . [0,1,2,1,3] [1,1,1,1,1]	ϕ	99.	1 n1 2 n1 3 n1 .
[0,1,2,3] [0,1,1,2,3]	ϕ	99	
[0, 1,2,3] .	1	1	
[ ]			
[1, 2 ,3]	20	99	

Are you combining the properties? You may need to use judgment to choose among the many combinations, or you could end up with *many* test cases.

lst	n1	n2	Purpose

3. Here is an example showing how `pytest` can be used to implement the first test case shown in Question 2:

```
def test_insert_after_at_front() -> None:
    """Test insert_after with one occurrence of n1 at the front of lst.
    """
    input_list = [0, 1, 2, 3]
    insert_after(input_list, 0, 99)
    expected = [0, 99, 1, 2, 3]
    assert input_list == expected
```

Choose one of your own test cases from Question 2 and implement it in `pytest`.

input_lst = [0,1,1,2,3]

insert_after (input_list , 0 , 99)

4. Here is a Tournament class that records game outcomes and reports statistics. Method bodies are omitted.

```
class Tournament:
    """A sports tournament.

    === Attributes ===
    teams:
        The names of the teams in this tournament.
    team_stats:
        The history of each team in this tournament. Each key is a team name,
        and each value is a list storing two non-negative integers:
        the number of games played and the number won.
```

Our current way of storing the data

{'a': (~~0~~ ~~1~~ 2, ~~0~~ ~~1~~ 2),
'b': (~~0~~ ~~1~~ 2, ~~0~~ 1),
'c': (~~0~~ ~~1~~ 2, 0)}

```
    === Sample usage ===

    >>> t = Tournament(['a', 'b', 'c'])
    >>> t.record_game('a', 'b', 10, 4)
    >>> t.record_game('a', 'c', 5, 1)
    >>> t.record_game('b', 'c', 2, 0)
    >>> t.best_percentage()
    'a'
    """
    # Attribute types
    teams: List[str]
    team_stats: Dict[str, List[int]]
```

One idea

{'a': [(10,4), (5,1)],
'b': [(4,10), (2,0)],
'c': [(1,5), (0,2)]}

something better?

```
    def __init__(self, teams: List[str]) -> None:
        """Initialize a new Tournament among the given teams.

        Note: Does not make an alias to <teams>.
        """

    def record_game(self, team1: str, team2: str,
                    score1: int, score2: int) -> None:
        """Record the fact that <team1> played <team2> with the given scores.

        <team1> scored <score1> and <team2> scored <score2> in this game.

        Precondition: team1 and team2 are both in this tournament.
        """

    def best_percentage(self) -> str:
        """Return the team name with the highest percentage of games won.

        If no team has won a game, return the empty string.
        Otherwise if there is a tie for best percentage, return the name of any
        of the tied teams.
        """
```

will have to change method bodies

(a) Are the instance attributes sufficient in order to implement method best_percentage? Explain.

Yes

(b) Identify another statistic that could be reported and for which the instance attributes are insufficient. How would you change the instance attributes to support it?

anything to do with ties, or goals

(c) What negative consequences might ensue if you changed the instance attributes?

See the fanclub demo + posted code.
(Also a quercus announcement about it.)