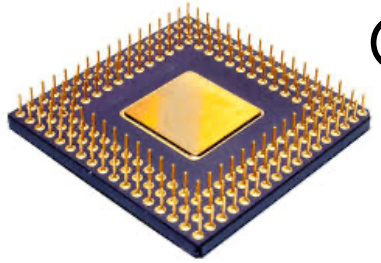
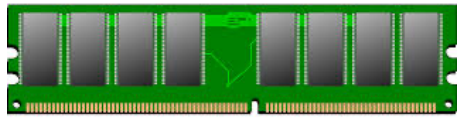
What is programming?

- Writing instructions for computers to perform tasks



CPU

Execute instructions



Memory

Data
Instructions

Instructions

- operators
- for loops
- if-else conditionals
- functions

...

Data

- lists
- tuples
- dictionaries
- sets

...

What is happening?

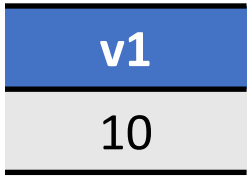
- When copying variables with basic types,

v1
10

```
v1 = 10
```

What is happening?

- When copying variables with basic types,



Object

v1 = 10

What is happening?

- When copying variables with basic types, the entire object is duplicated.

v1
10

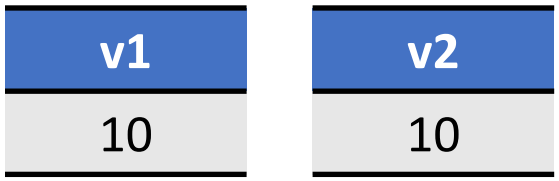
v2
10

Object

```
v2 = v1
```

What is happening?

- When copying variables with basic types, the entire object is duplicated.



Object

New Object

```
v2 = v1
```

What is happening?

- When copying variables with basic types, the entire object is duplicated. So, changing one does not affect the other.

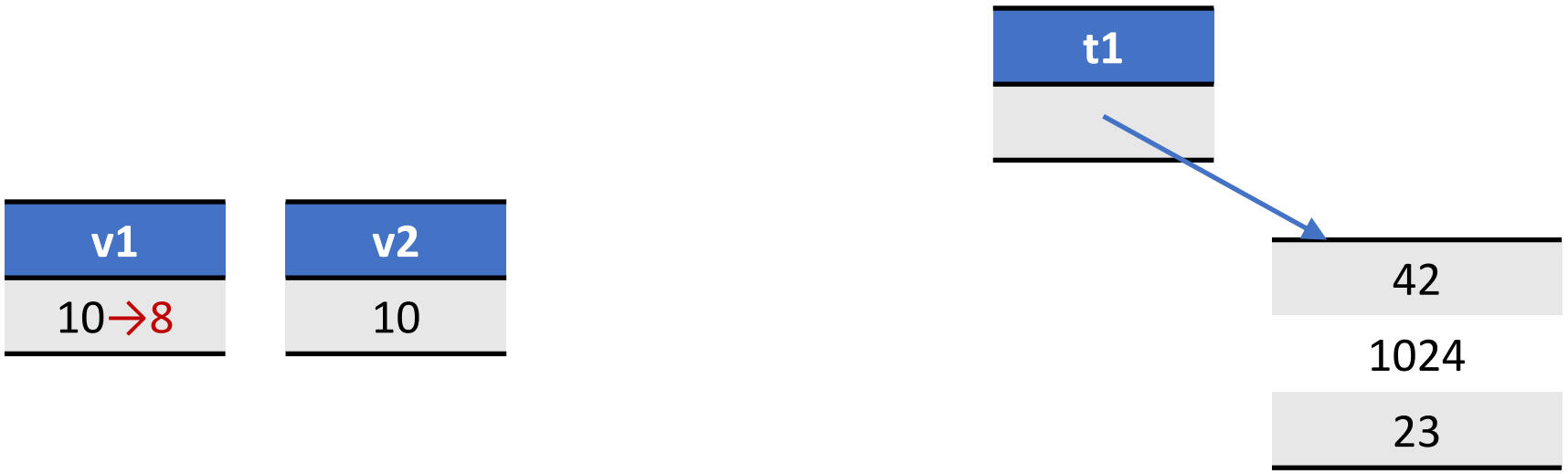
v1
10→8

v2
10

```
v1 = 8
```

What is happening?

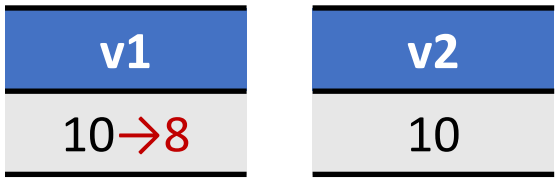
- When copying variables with basic types, the entire object is duplicated. So, changing one does not affect the other.
- When copying lists,

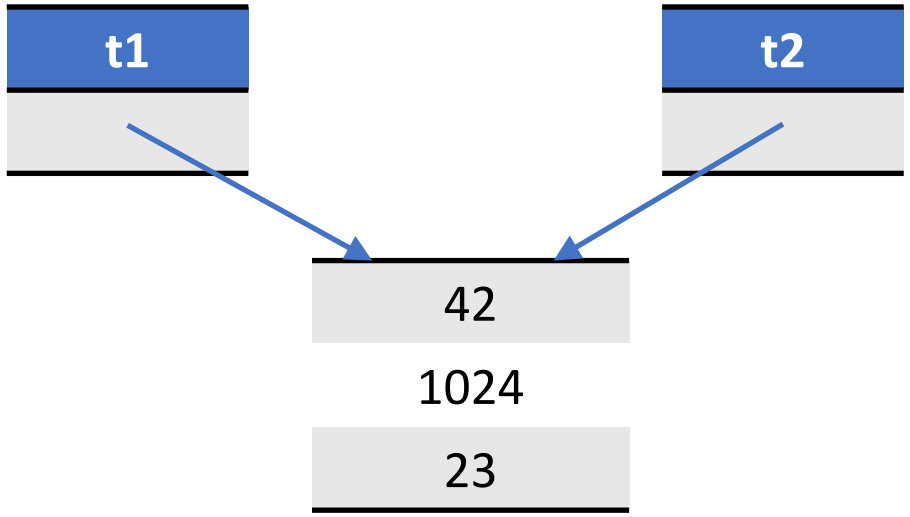



```
t1 = [42, 1024, 23]
```

What is happening?

- When copying variables with basic types, the entire object is duplicated. So, changing one does not affect the other.
- When copying lists, it copies only its address.

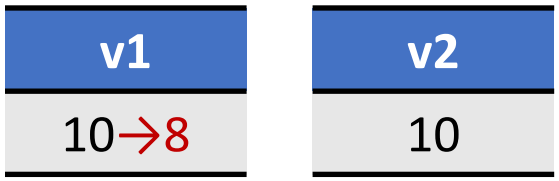


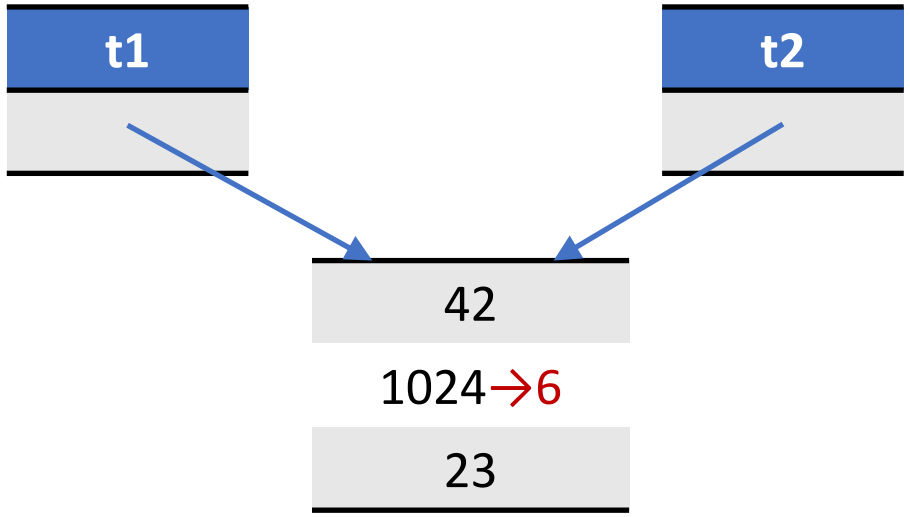



```
t2 = t1
```

What is happening?

- When copying variables with basic types, the entire object is duplicated. So, changing one does not affect the other.
- When copying lists, it copies only its address. So, changing one affects the other.






```
t1[1] = 6
```

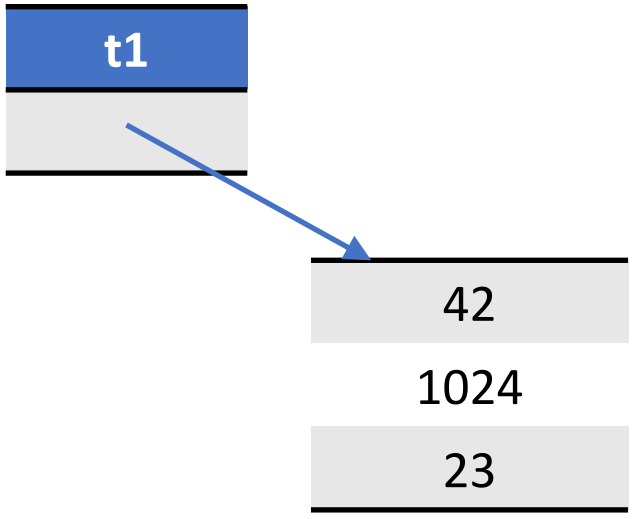
How can I make a real copy of a list?

`list.copy()` makes a shallow copy of a list

```
t1 = [42, 1024, 23]  # create a list
t2 = t1              # this only copies the address (copy by reference)
t3 = t1.copy()       # list.copy() function actually makes a copy
t1[1] = 6            # now let's modify t1.
print(t1)            # t1 is already modified
print(t2)            # would t2 be also modified?
print(t3)            # would t3 be also modified?
```

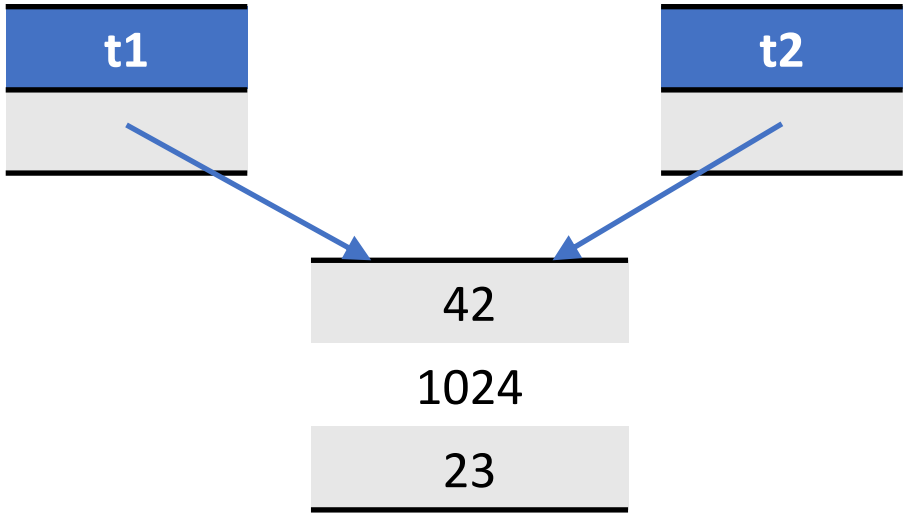
```
[42, 6, 23]
[42, 6, 23]
[42, 1024, 23]
```

What is happening in shallow copy?




```
t1 = [42, 1024, 23]
```

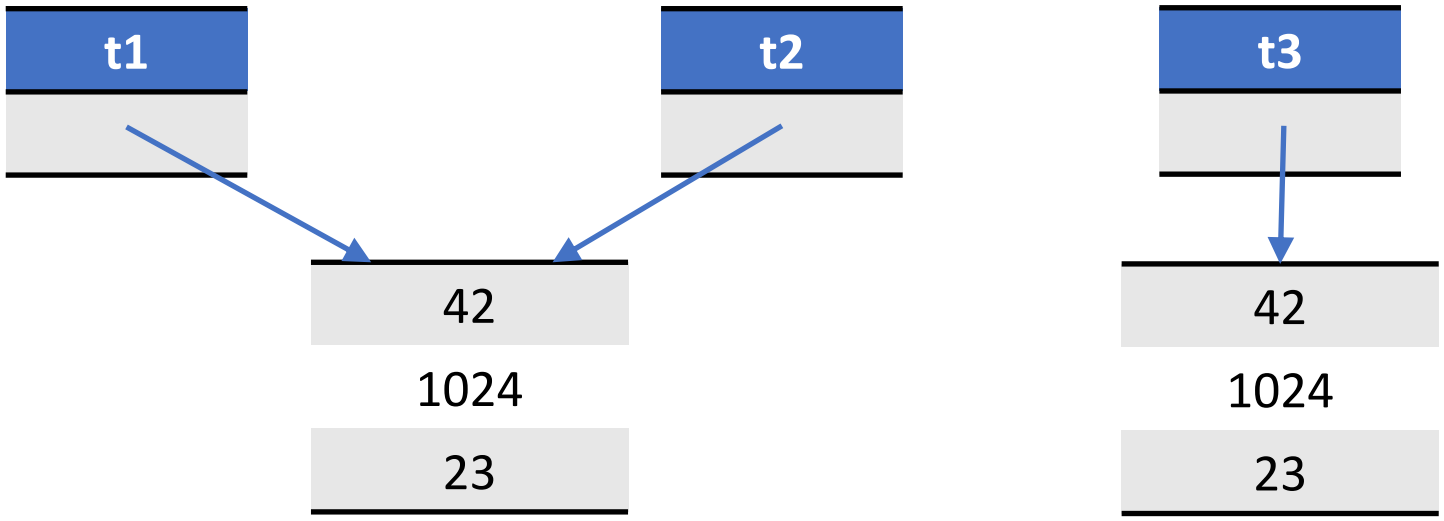
What is happening in shallow copy?




```
t2 = t1
```

What is happening in shallow copy?

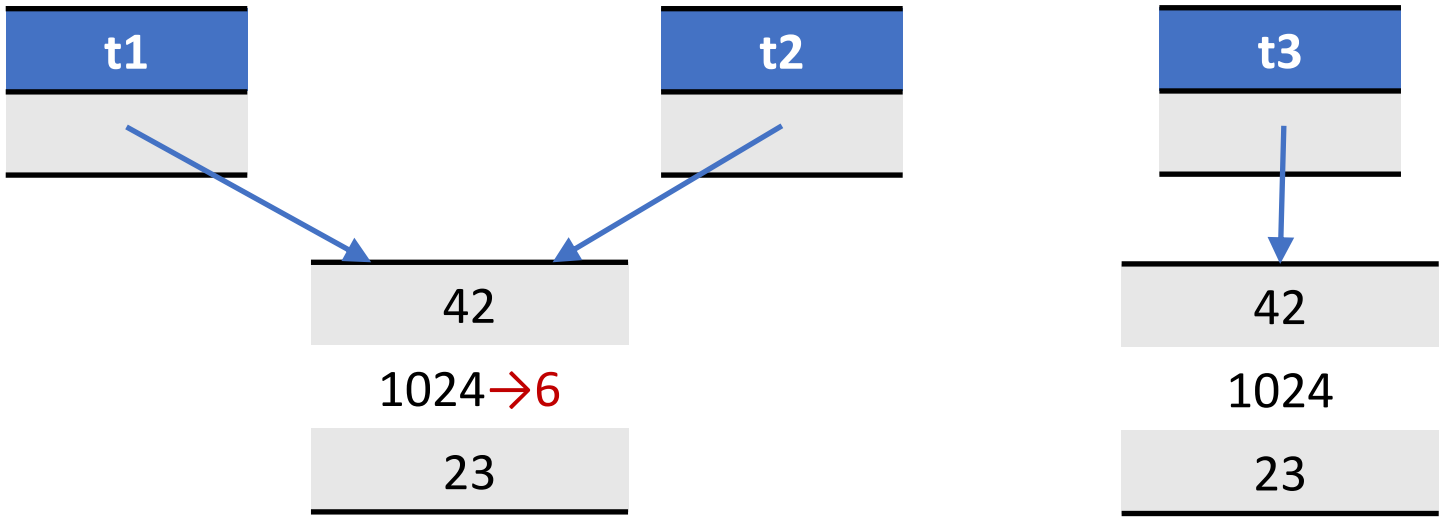
- When **`copy()`** function is called, it copies the list itself.




```
t3 = t1.copy()
```

What is happening in shallow copy?

- When **`copy()`** function is called, it copies the list itself.




```
t1[1] = 6
```

A nested sequence

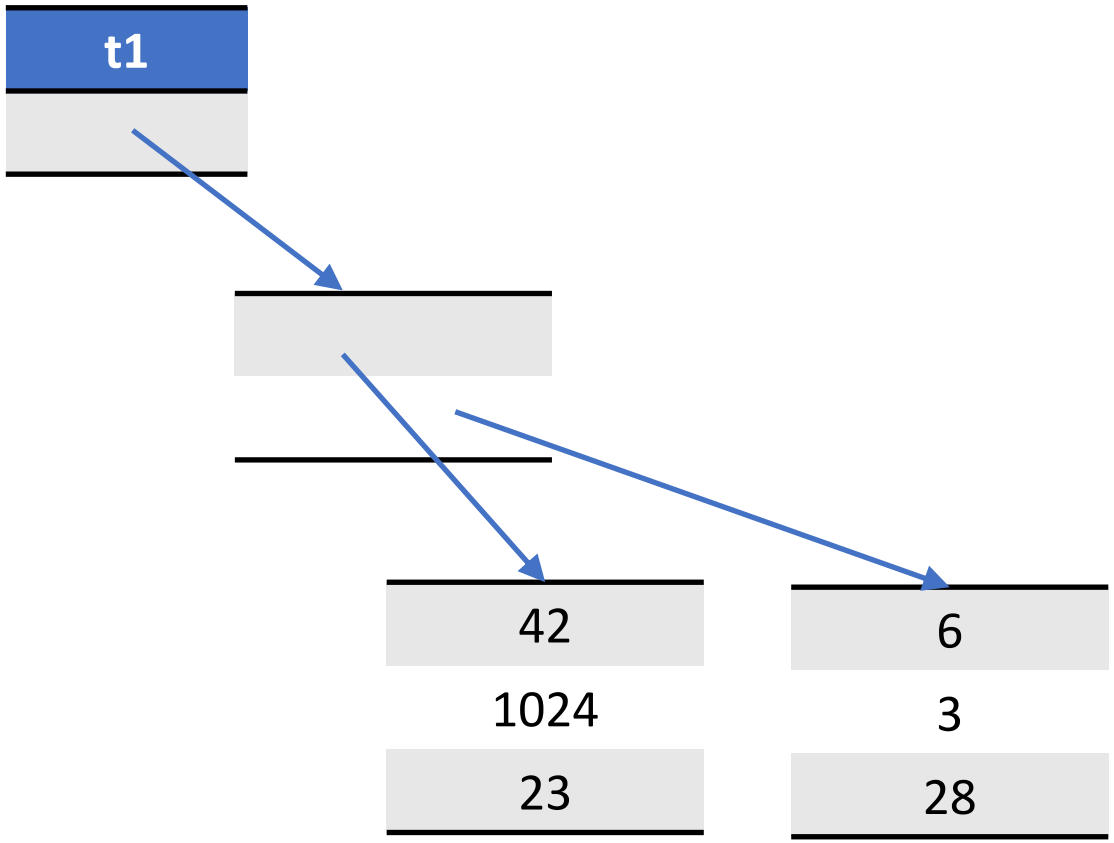
- You may create a sequence of a sequence hierarchically.
- Nested data structure can be made in multiple levels.

```
# create a list of list
names = [['Hyun', 'Kang'], ['Bharmar','Mukherjee'], ['Mike', 'Boehnke']]
print(names)
print(names[1])    # 2nd element is still a list
print(names[1][1]) # access the actual element using double index
```

```
[['Hyun', 'Kang'], ['Bharmar', 'Mukherjee'], ['Mike', 'Boehnke']]
['Bharmar', 'Mukherjee']
Mukherjee
```

What is happening in shallow vs. deep copy?

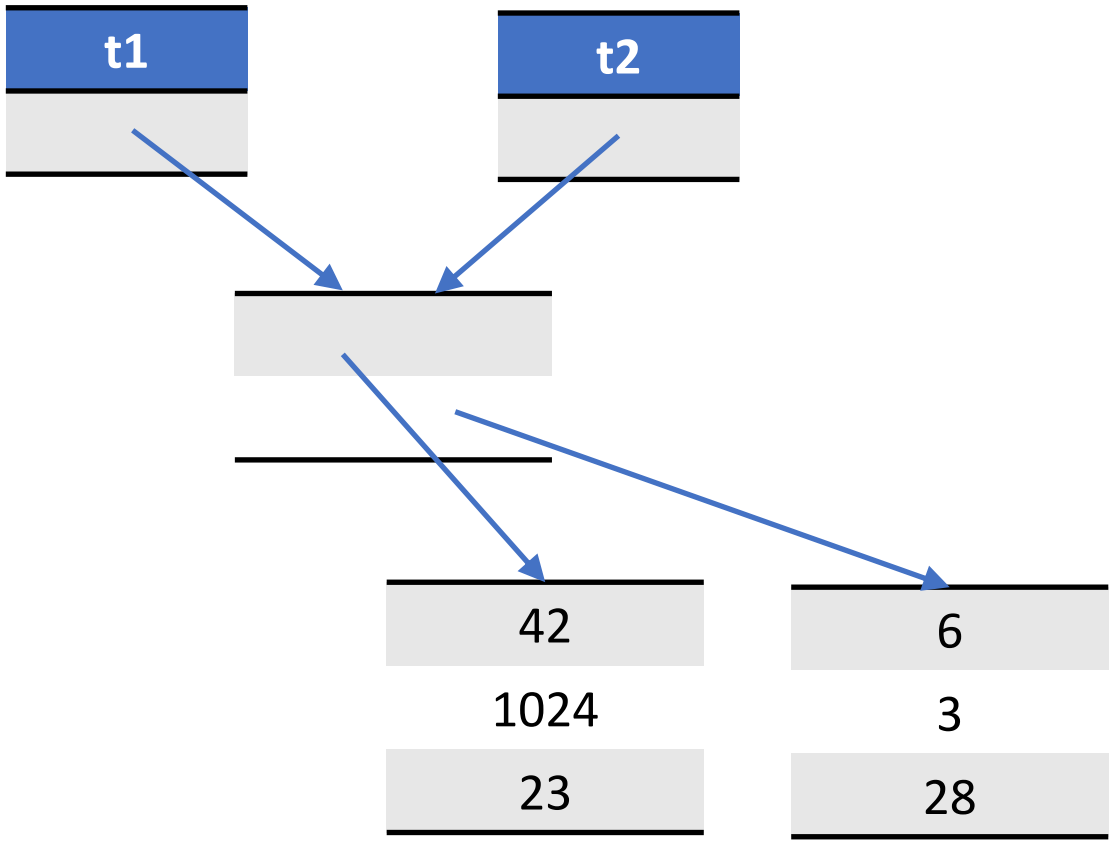
- Created a list of list (i.e. nested list)




```
t1 = [ [42,1024,23], [6,3,28] ]
```

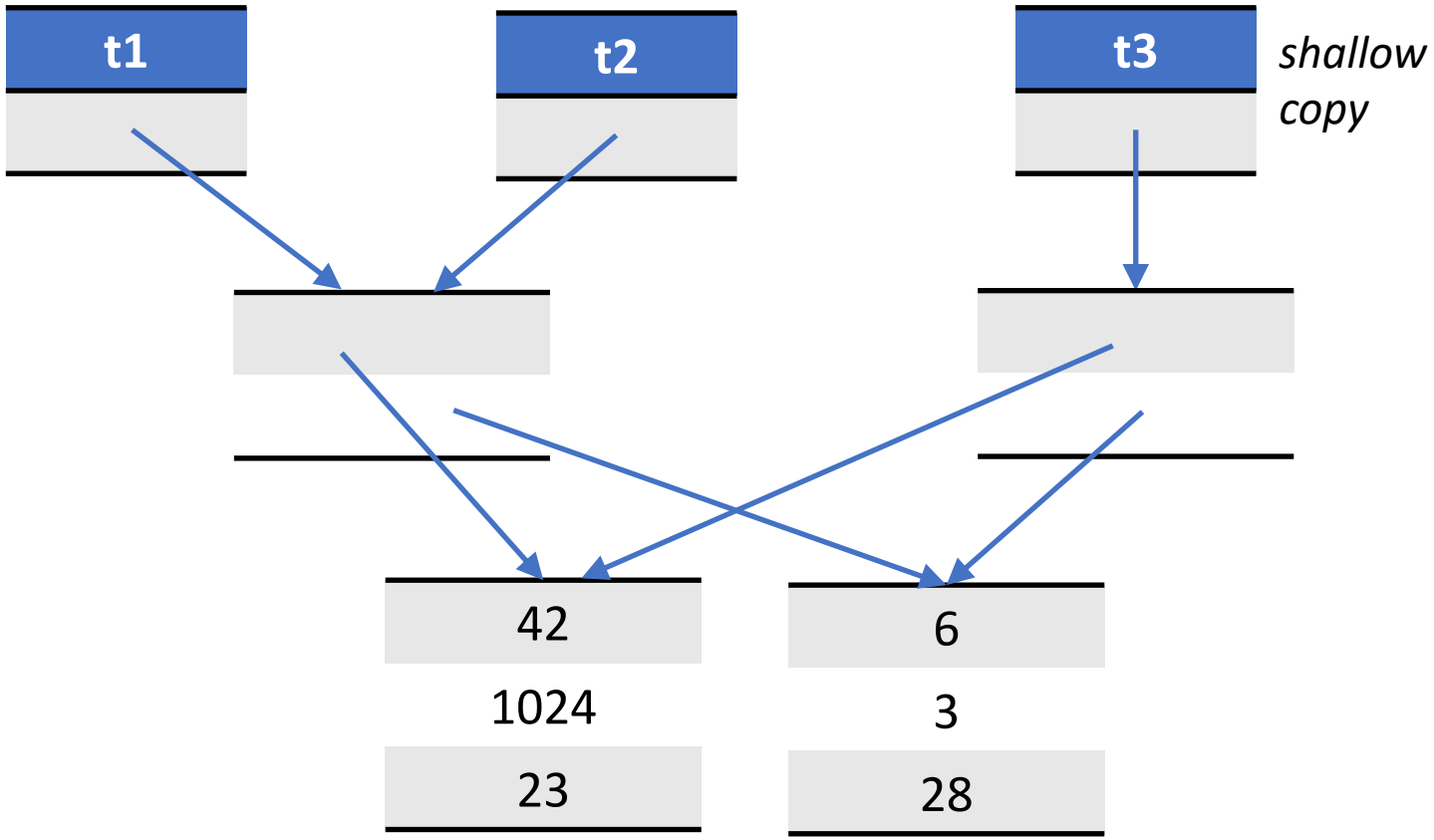
What is happening in shallow vs. deep copy?

- `t2 = t1` only copies the address.



What is happening in shallow vs. deep copy?

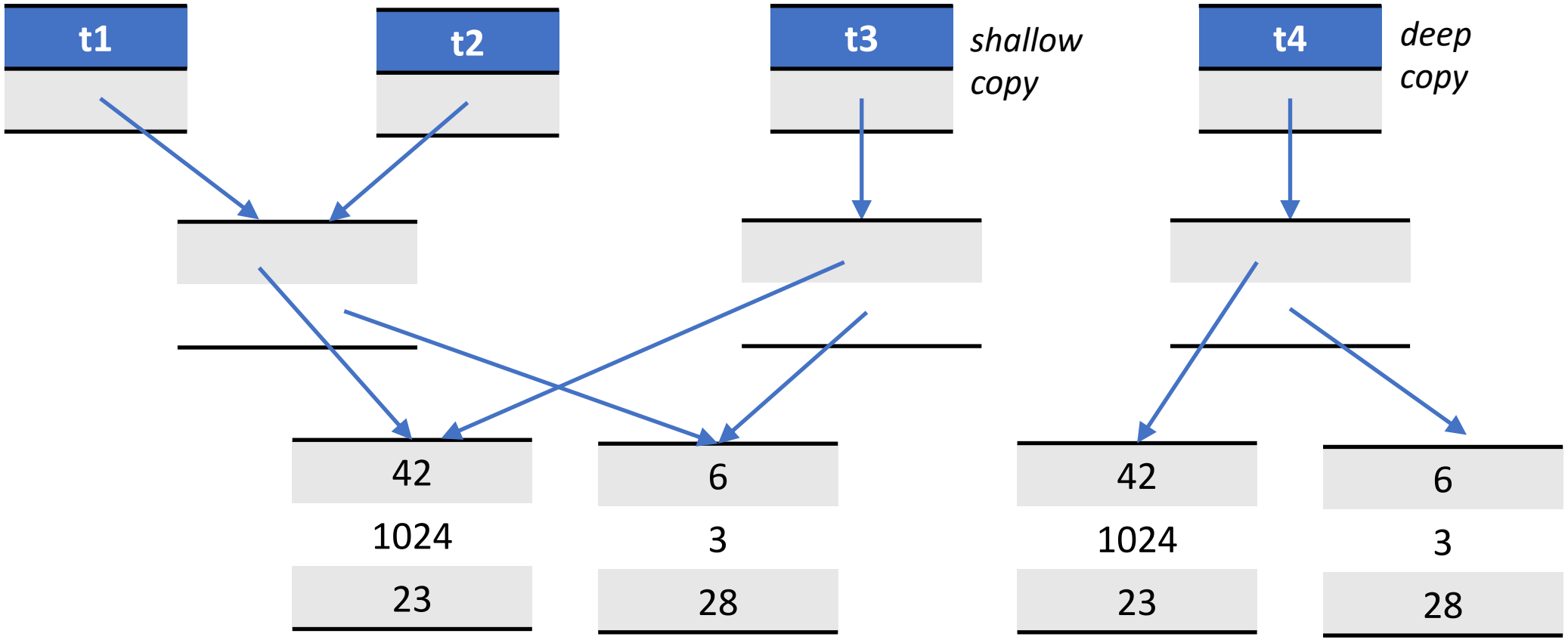
- **`t1.copy()`** only copies the list that **`t1`** directly refers to.



`t3 = t1.copy()`

What is happening in shallow vs. deep copy?

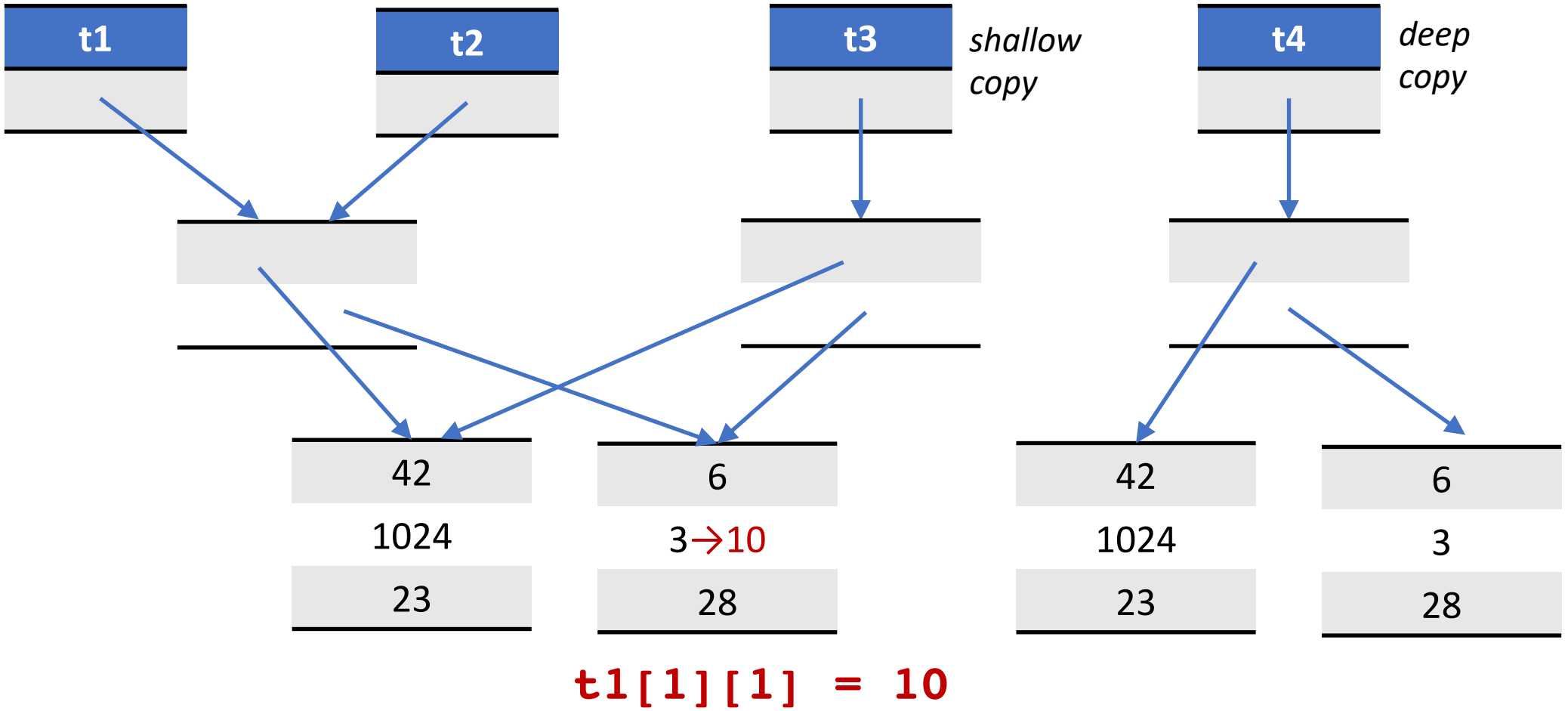
- **`copy.deepcopy()`** follows all references (i.e., addresses) and make a whole copy.




```
t4 = copy.deepcopy(t1)
```

What is happening in shallow vs. deep copy?

- **`copy.deepcopy()`** follows all references (i.e., addresses) and make a whole copy.



Shallow vs. Deep copy

- If **`.copy()`** function makes a shallow copy, is there something called deep copy?

```
import copy  # copy.deepcopy function performs a deep copy
t1 = [ [42, 1024, 23], [6, 3, 28] ] # create a nested list
t2 = t1                             # copy by reference
t3 = t1.copy()                      # shallow copy
t4 = copy.deepcopy(t1)              # deep copy
t1[1][1] = 10                       # modified an element
print(t1)                           # t1 should be modified already
print(t2)                           # would t2 be modified?
print(t3)                           # how about t3?
print(t4)                           # how about t4?
```

```
[[42, 1024, 23], [6, 10, 28]]
[[42, 1024, 23], [6, 10, 28]]
[[42, 1024, 23], [6, 10, 28]]
[[42, 1024, 23], [6, 3, 28]]
```

Be mindful about copy when passing arguments

```
def double(x):   # Function that doubles all elements
  for i in range(len(x)):
    x[i] = 2 * x[i]
  return x

t1 = [42, 1024, 23]
t2 = double(t1)
t3 = double(t1.copy())
print(t1, t2, t3, sep='\n')
t2[2] = 6
t3[2] = 28
print(t1, t2, t3, sep='\n')
```

```
[84, 2048, 46]
[84, 2048, 46]
[168, 4096, 92]
[84, 2048, 6]
[84, 2048, 6]
[168, 4096, 28]
```

Is this a satisfactory outcome?

A very simple modification

```
def double(x):    # Function that doubles all elements
  x = x.copy()    # Make a local copy of it
  for i in range(len(x)):
    x[i] = 2 * x[i]
  return x        # Now returning a copy

t1 = [42, 1024, 23]
t2 = double(t1)
t3 = double(t1.copy())
print(t1, t2, t3, sep='\n')
t2[2] = 6
t3[2] = 28
print(t1, t2, t3, sep='\n')
```

```
[42, 1024, 23]
[84, 2048, 46]
[84, 2048, 46]
[42, 1024, 23]
[84, 2048, 6]
[84, 2048, 28]
```

Does this look better?