What the f*ck Python! 🐍

is is not what it is!

```python
a = 256
b = 256
```

```python
a is b
# ?
```

```python
a = 257
b = 257
```

```python
a is b
# ?
```

```python
a = 257; b = 257
a is b
# ?
```

The difference between is and ==

- is operator checks if both the operands refer to the same object (i.e., it checks if the identity of the operands matches or not).

- == operator compares the values of both the operands and checks if they are the same.

- So is is for reference equality and == is for value equality.

- When you start up python the numbers from -5 to 256 will be allocated. These numbers are used a lot, so it makes sense just to have them ready.
- When a and b are set to 257 in the same line, the Python interpreter creates a new object, then references the second variable at the same time. If you do it on separate lines, it doesn't "know" that there's already 257 as an object.
- It's a compiler optimization and specifically applies to the interactive environment. When you enter two lines in a live interpreter, they're compiled separately, therefore optimized separately. If you were to try this example in a .py file, you would not see the same behavior, because the file is compiled all at once.

Strings can be tricky sometimes

```python
a = 'some_string'
id(a)
id('some' + '_' + ‘string')
```

```python
a = 'wtf'
b = 'wtf'
a is b
# ?
```

```python
a = 'wtf!'
b = 'wtf!'
a is b
# ?
```

```python
a, b = 'wtf!', 'wtf!'
a is b
# ?
```

```python
'a' * 20 is 'aaaaaaaaaaaaaaaaaaaa'
'a' * 21 is 'aaaaaaaaaaaaaaaaaaaaa'  # python2/python3
```

String interning

- All length 0 and length 1 strings are interned
- Strings are interned at compile time(‘wtf’ vs ‘’.join(‘w’, ’t’, ‘f’))
- Strings that are not composed of ASCII letters are not interned
- When a and b are set to ‘wtf!’ in the same line, the Python interpreter creates a new object, then references the second var at the same time
- Constant folding(peephole optimization) only occurs for strings having length less than or equal 20(Python2)

Time for some hash brownies!

```
some_dict = {}
some_dict[5.5] = "Ruby"
some_dict[5.0] = "JavaScript"
some_dict[5] = "Python"
```

```
some_dict[5.5]
# ?
some_dict[5.0]
# ?
some_dict[5]
# ?
```

"Python" destroyed the existence of "JavaScript"?

- Python dictionaries check for equality and compare the hash value to determine if two keys are the same
- Immutable objects with same value always have the same hash in Python
- When the statement some_dict[5] = "Python" is executed, the existing value "JavaScript" is overwritten with "Python" because Python recognizes 5 and 5.0 as the same keys of the dictionary some_dict

Return return everywhere!

```python
def some_func():
    try:
        return 'from_try'
    finally:
        return 'from_finally'


some_func()
# ?
```

- When a return, break or continue statement is executed in the try suite of a "try…finally" statement, the finally clause is also executed ‘on the way out

- The return value of a function is determined by the last return statement executed. Since the finally clause always executes, a return statement executed in the finally clause will always be the last one executed

Deep down, we're all the same

```python
class WTF:
    pass


WTF() == WTF()  # two different instances can't be equal
# ?

WTF() is WTF()  # identities are also different
# ?

hash(WTF()) == hash(WTF()) # hashes _should_ be different as well
# ?

id(WTF()) == id(WTF())
# ?
```

•

- When id was called, Python created a WTF class object and passed it to the id function. The id function takes its id (its memory location), and throws away the object. The object is destroyed.

- When we do this twice in succession, Python allocates the same memory location to this second object as well. Since (in CPython) id uses the memory location as the object id, the id of the two objects is the same.

- So, object's id is unique only for the lifetime of the object. After the object is destroyed, or before it is created, something else can have the same id.

- But why did the is operator evaluated to False?

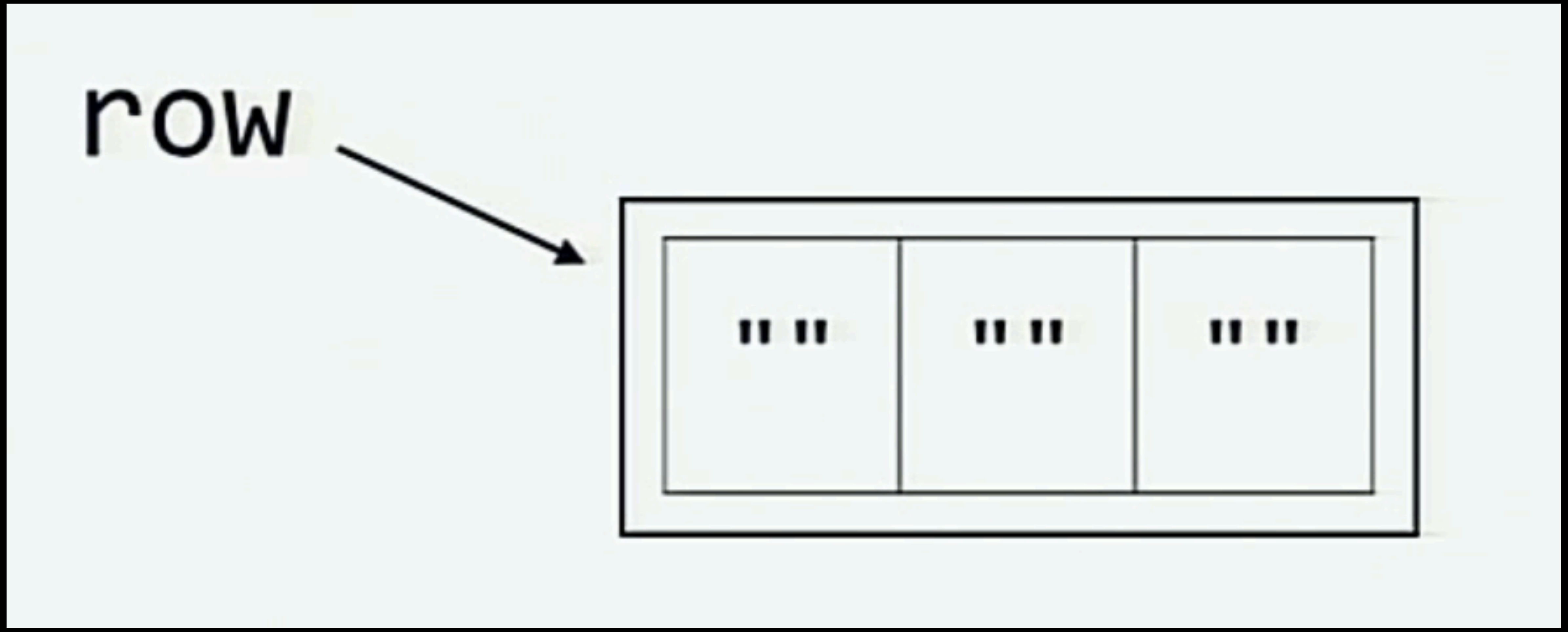
A tic-tac-toe where X wins in the first attempt!

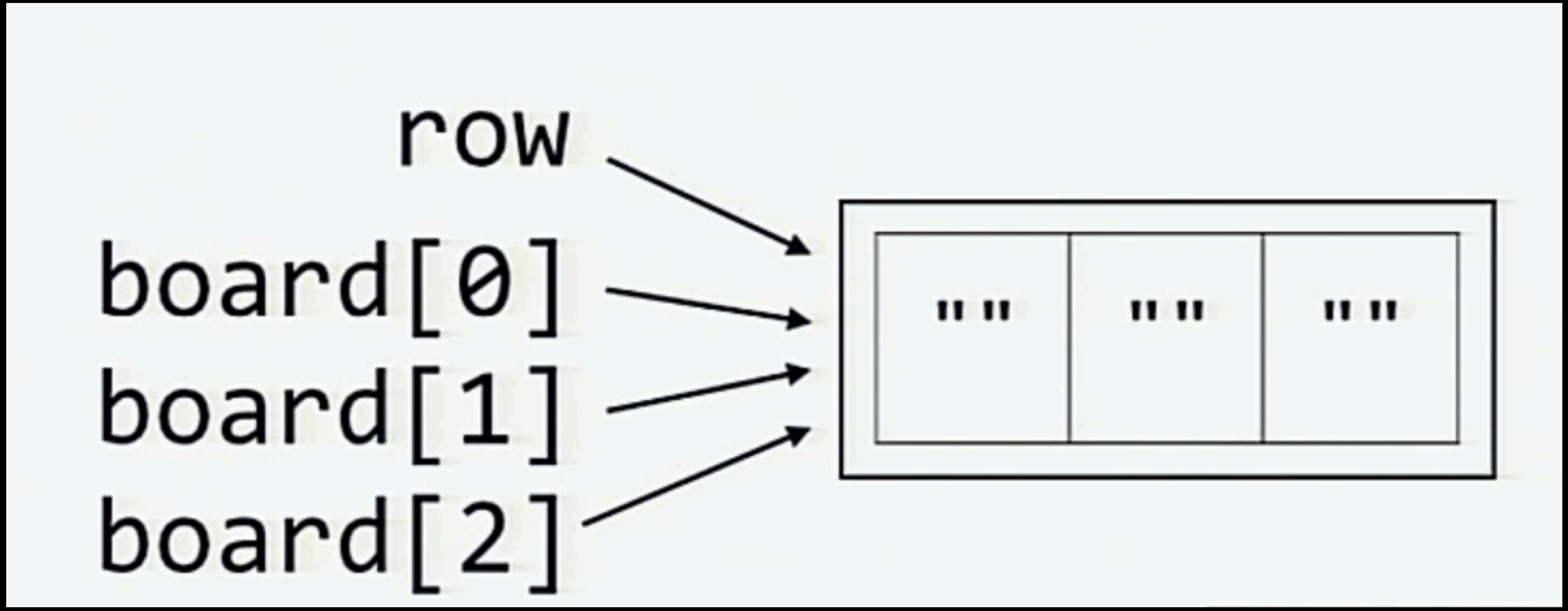
```
# Let's initialize a row
row = [""] * 3 #row i['', '', '']
# Let's make a board
board = [row] * 3


board
board[0]
board[0][0]

board[0][0] = "X"
board
# ?
```

- And when the board is initialized by multiplying the row, this is what happens inside the memory (each of the elements board[0], board[1] and board[2] is a reference to the same list referred by row)





- You can find more here: https://github.com/satwikkansal/wtfpython.git