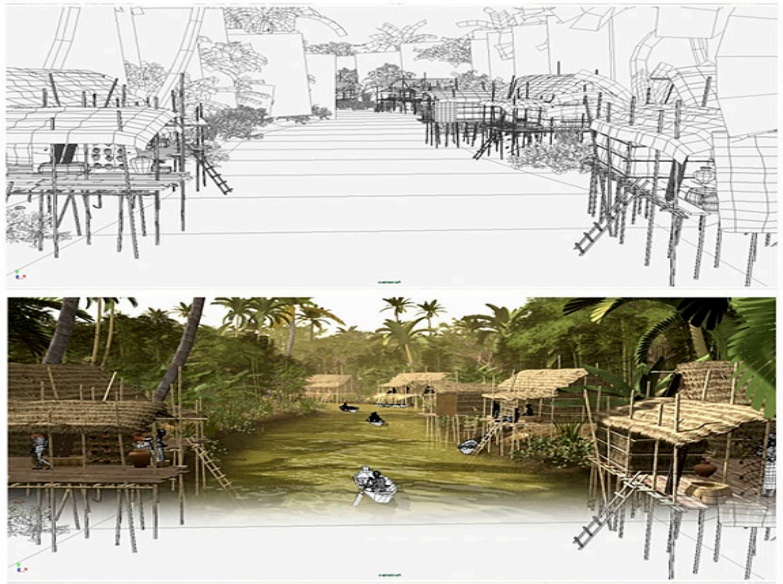
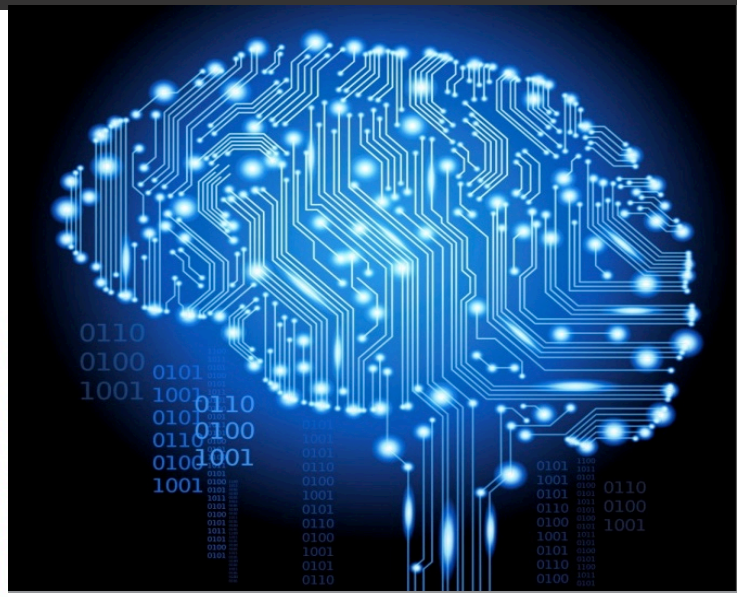
MONASH University

Information Technology

FIT1008/2085
Namespaces, Scoping and Mutability

Prepared by:
Maria Garcia de la Banda
Revised by D. Albrecht, J. Garcia





Objectives for this lesson

- **To learn how OO in Python affects:**
  - Variable and value creation, variable assignment and aliasing
- **To learn about:**
  - Names and Namespaces
  - Scopes and Scoping Rules
  - Mutable and immutable objects

  and how are defined in Python, particularly in the context of OO
- **To be able to follow Python code involving**
  - Variable assignments and aliasing in the OO context
  - Mutable and immutable objects

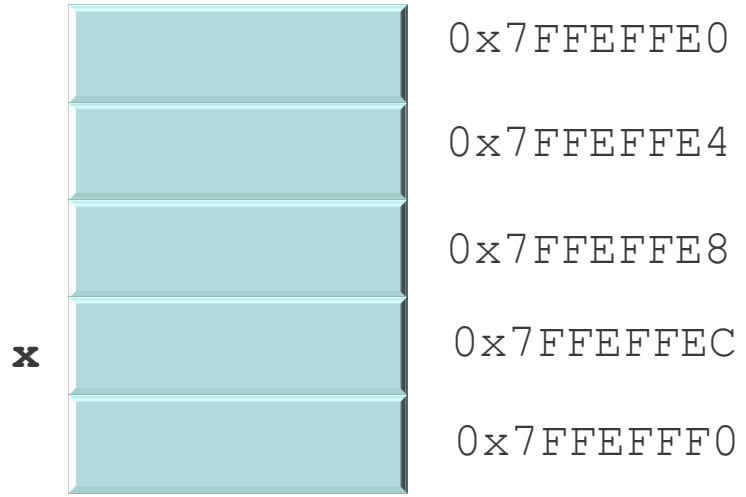
MONASH University

Python Variables and Objects

Variable representation

You saw this in MIPS: global variables are labels (which are addresses in the data segment), and locals are directly addresses in the runtime stack like -4($fp)

- **What is a program variable?**
  - A name (identifier) of some "something"
- **The name in almost all languages refers to a memory location**
  - The something depends on what you assign to it
- **As seen in MIPS, memory is divided into portions**
  - bits, bytes, words…
- **Each portion has an address**
- **Internally, a variable `x` is an address**
  - Say `0x7FFEFFEC`
- **That memory address contains…?**
  - The "something"?



Variable representation in Python

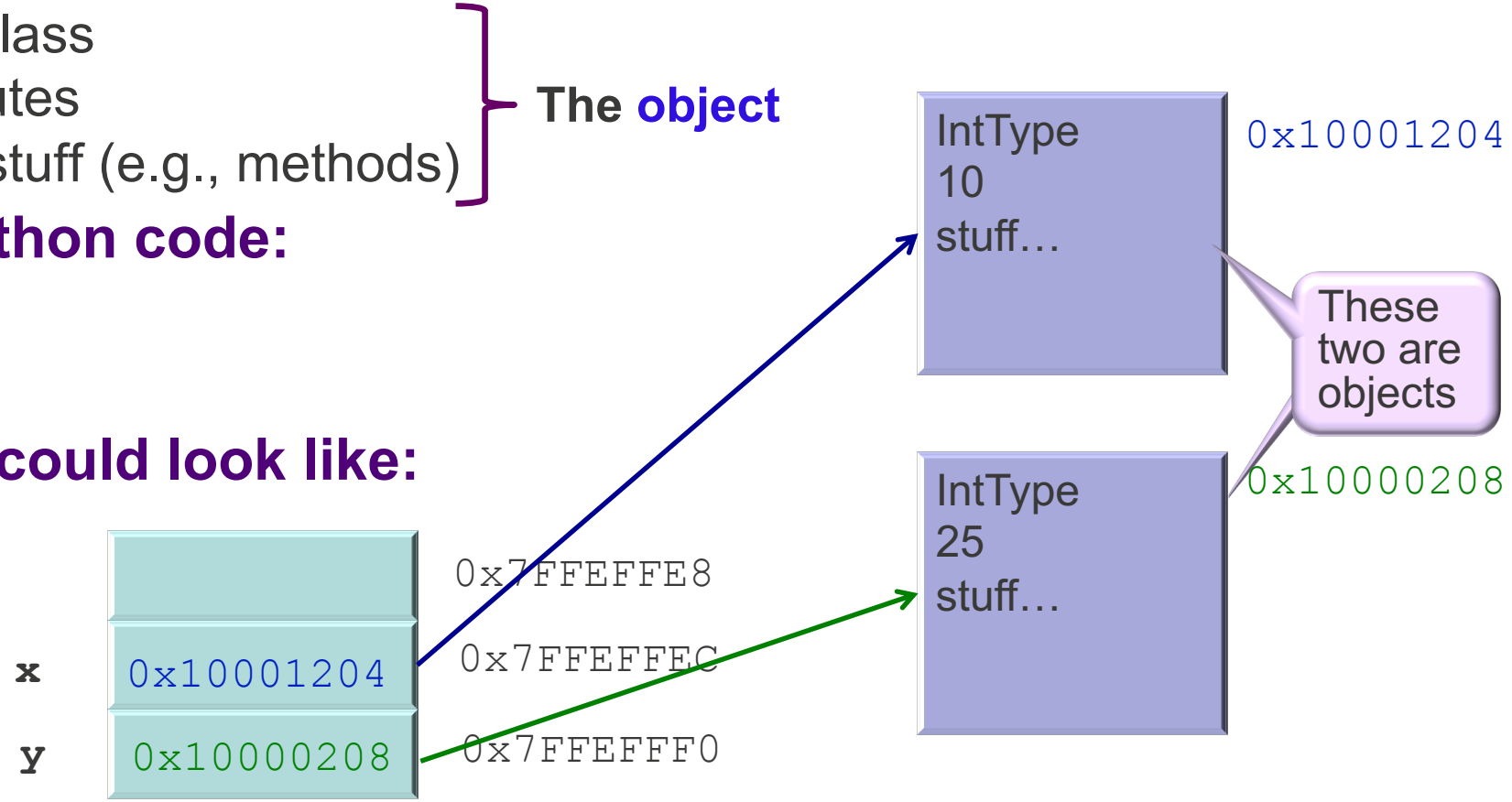
Like we treated arrays in MIPS: pointer to the array located in the data segment

- **The content depends … on the language!**
- **In Python: it is a reference to the memory location containing an object**
  - The object's class
  - Its data attributes
  - Lots of other stuff (e.g., methods)

  } **The object**
- **Consider the Python code:**

```
x = 10
y = 25
```

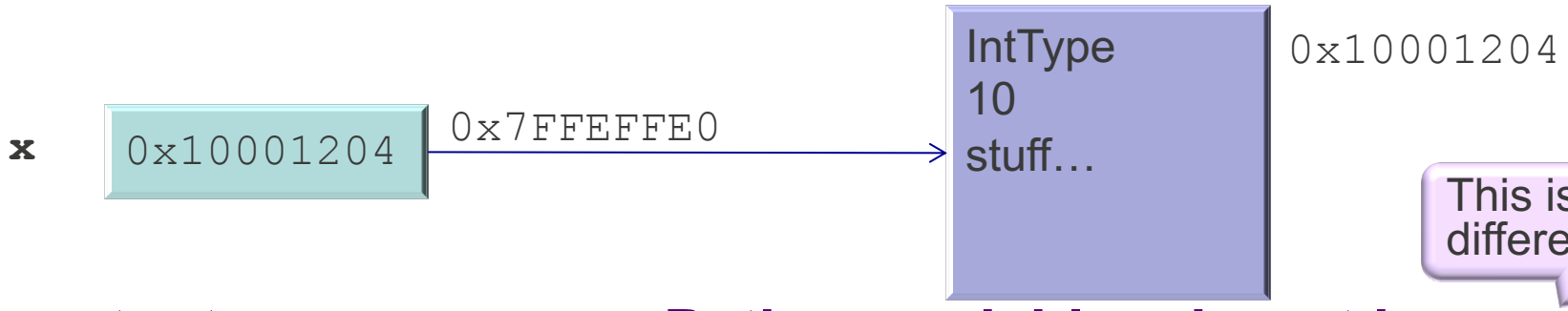
- **In memory, this could look like:**



Creating variables in Python

- **A variable is created when you first assign it a value**
  - In many other languages, variables can be *declared*
    - This means created *without* a value (or given a default); not possible in Python
- **So when you say `x = 10` in Python, it:**
  1. Creates an object to represent `10`, starting at some address
  2. Creates the variable `x` if it does not exist already
  3. Links it with the object created (assigns the address to `x`)

"Under the Python hood" `x = 10` is "syntactic sugar" for something like `x = Integer(10)`



This is why you can assign values of different types to a Python variable

- **Important consequence: Python variables do not have a type**
  - Types are associated with values (i.e., with objects)

Our visualisation of objects in Python

- **To simplify things, we will:**
  - Only display values within the object
    - The type and other stuff will be ignored for now
  - Ignore the exact value of the references (i.e., the address)
    - We will use arrows to represent them
- **So `x = 10` will look like:**

- **I will keep `x` outside the box to emphasize the content is the reference (as in MIPS for arrays), not the name**

Assigning variables to other values

- **We said: in Python variables are always references to objects**
  - Changing the assignment does not alter the object itself
  - It only alters the reference: the variable will refer (point) to a different object
- **Consider the code:**

```
x = 10
x = x + 3
```

- **Let's see how it executes:**
  1. Creates object `10` somewhere

Assigning variables to other values

- **We said: in Python variables are always references to objects**
  - Changing the assignment does not alter the object itself
  - It only alters the reference: the variable will refer (point) to a different object
- **Consider the code:**

```
x = 10
x = x + 3
```

- **Let's see how it executes:**
  1. Creates object `10` somewhere
  2. Creates variable `x`

Assigning variables to other values

- **We said: in Python variables are always references to objects**
  - Changing the assignment does not alter the object itself
  - It only alters the reference: the variable will refer (point) to a different object
- **Consider the code:**

```
x = 10
x = x + 3
```

- **Let's see how it executes:**
  1. Creates object `10` somewhere
  2. Creates variable `x`
  3. Links `x` to `10`

Assigning variables to other values

- **We said: In Python variables are always references to objects**
  - Changing the assignment does not alter the object itself
  - It only alters the reference: the variable will refer (point) to a different object
- **Consider the code:**

```
x = 10
x = x + 3
```

- **Let's see how it executes:**
  1. Creates object `10` somewhere
  2. Creates variable `x`
  3. Links `x` to `10`
  4. Evaluates `x+3`

A variable in an expression is immediately replaced with the object it currently refers to. Then the expression is evaluated.

This is why you MUST assign a value to a variable before using it in Python

Assigning variables to other values

- **We said: in Python variables are always references to objects**
  - Changing the assignment does not alter the object itself
  - It only alters the reference: the variable will refer (point) to a different object
- **Consider the code:**

```
x = 10
x = x + 3
```

- **Let's see how it executes:**
  1. Creates object `10` somewhere
  2. Creates variable `x`
  3. Links `x` to `10`
  4. Evaluates `x+3`
  5. Creates object `13`

A variable in an expression is immediately replaced with the object it currently refers to. Then the expression is evaluated.

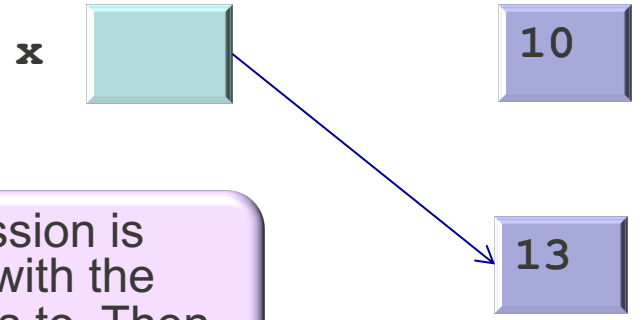
This is why you MUST assign a value to a variable before using it in Python

Assigning variables to other values

- **We said: In Python variables are always references to objects**
  - Changing the assignment does not alter the object itself
  - It only alters the reference: the variable will refer (point) to a different object
- **Consider the code:**

```
x = 10
x = x + 3
```

- **Let's see how it executes:**
  1. Creates object `10` somewhere
  2. Creates variable `x`
  3. Links `x` to `10`
  4. Evaluates `x+3`
  5. Creates object `13`
  6. Links `x` to `13`



What happens to this object?

The same as "deallocated" arrays in the heap: they become "garbage" unless some other variable is still pointing to them

A variable in an expression is immediately replaced with the object it currently refers to. Then the expression is evaluated.

This is why you MUST assign a value to a variable before using it in Python

Assigning variables to other variables (aliasing)

- **We have seen that every time a constant appears in our program:**
  - Python creates a new object (a chunk of memory somewhere) to represent it
- **What about assigning a variable to another variable? Consider:**

```
x = 10
y = x
x = 'hi'
```

If I print `x` and `y` after this, what would it say?

- **Let's see how it executes:**
  1. Creates object `10` somewhere

Assigning variables to other variables (aliasing)

- **Thus, every time a constant appears in our program:**
  - Python creates a new object (a chunk of memory somewhere) to represent it
- **What about assigning a variable to another variable? Consider:**

```
x = 10
y = x
x = 'hi'
```

If I print `x` and `y` after this, what would it say?

- **Let's see how it executes:**
  1. Creates object `10` somewhere
  2. Creates variable `x`

Assigning variables to other variables (aliasing)

- **Thus, every time a constant appears in our program:**
  - Python creates a new object (a chunk of memory somewhere) to represent it
- **What about assigning a variable to another variable? Consider:**

```
x = 10
y = x
x = 'hi'
```

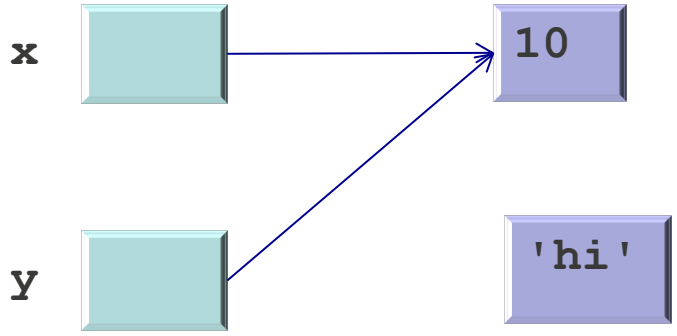
If I print `x` and `y` after this, what would it say?

- **Let's see how it executes:**
  1. Creates object `10` somewhere
  2. Creates variable `x`
  3. Links it to the object

Assigning variables to other variables (aliasing)

- **Now what? Time to execute `y = x`**
- **Variable `x` already exists, so no need to create it. So:**
    4. Creates variable `y`
    5. Links it to the object pointed to by `x`
- **Now time to execute `x = 'hi'`**
    4. Creates object `'hi'`
    5. Links `x` to this object

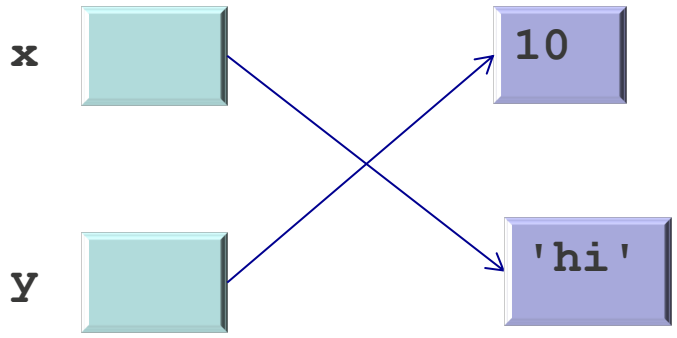
In MIPS: the code would load the value of x (which is an address) into, say, $t0 and then store it into y. So they have the same address and thus, the same link



Assigning variables to other variables (aliasing)

- **Now what? Time to execute `y = x`**
- **Variable `x` already exists, so no need to create it. So:**
    4. Creates variable `y`
    5. Links it to the object pointed to by `x`
- **Now time to execute `x = 'hi'`**
    4. Creates object `'hi'`
    5. Links `x` to this object
- **If we now print their values…**

```
>>> x = 10
>>> y = x
>>> x = 'hi'
>>> x;y
'hi'
10
```



And another example

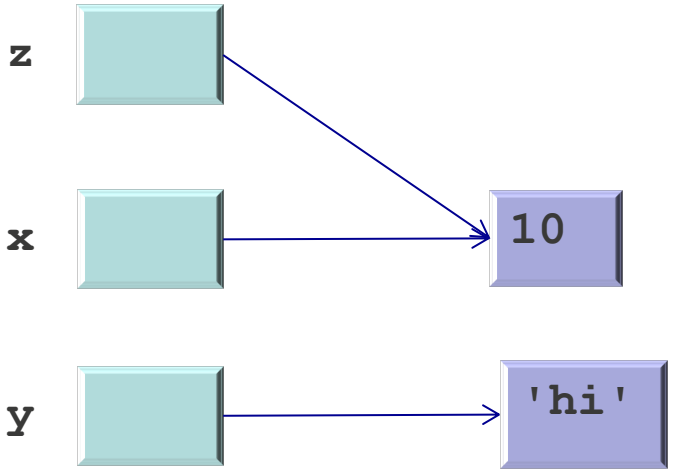
- **Consider the code**

```
>>> x = 10
>>> y = 'hi'
>>> z = x
>>> x = y
>>> x;y;z
```

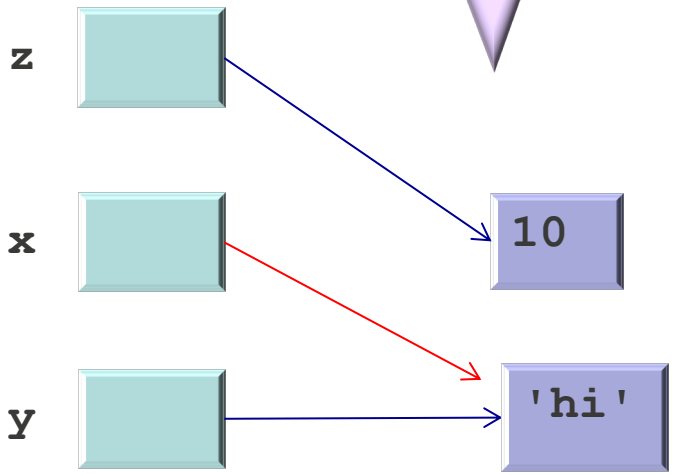
So it prints:

```
'hi'
'hi'
10
```

- **What will it print?**
- **After the first 3 lines it looks like:**



After the 4th line, it looks like this (only change in red):



Mixing assignments, aliasing and evaluations

- **Consider the code**

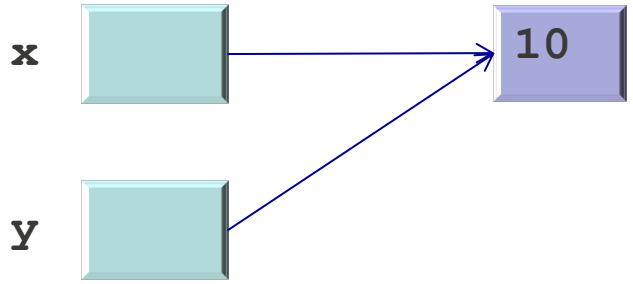
```
>>> x = 10
>>> y = x
>>> x = x+2
>>> x;y
```

Creates an object for the value resulting from evaluating the expression

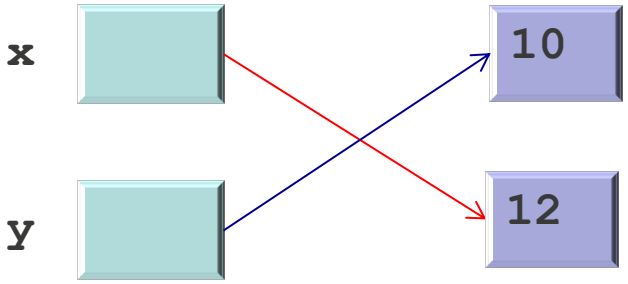
So it prints:

```
12
10
```

- **What will it print?**
- **After the first 2 lines it looks like:**



After the 3rd , it looks like this :



MONASH University

Names, Namespaces, Scopes and Scoping Rules

Names

- **We said that all values in Python are objects in the heap**
  - This includes functions, methods, modules, types… they are all objects!
- **All identifiers in Python are simply names that point to an object**
  - Like a MIPS variable with an address as value, pointing to a memory position
- **Careful when reusing names then; it can get very confusing:**

```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
>>>
```

```
>>> a_name = 'hello'
>>> a_name
'hello'
>>> class a_name:
...     i = 8
...
>>> a_name
<class '__main__.a_name'>
```

All the above code created a single variable name: `a_name`

Namespaces (or environments)

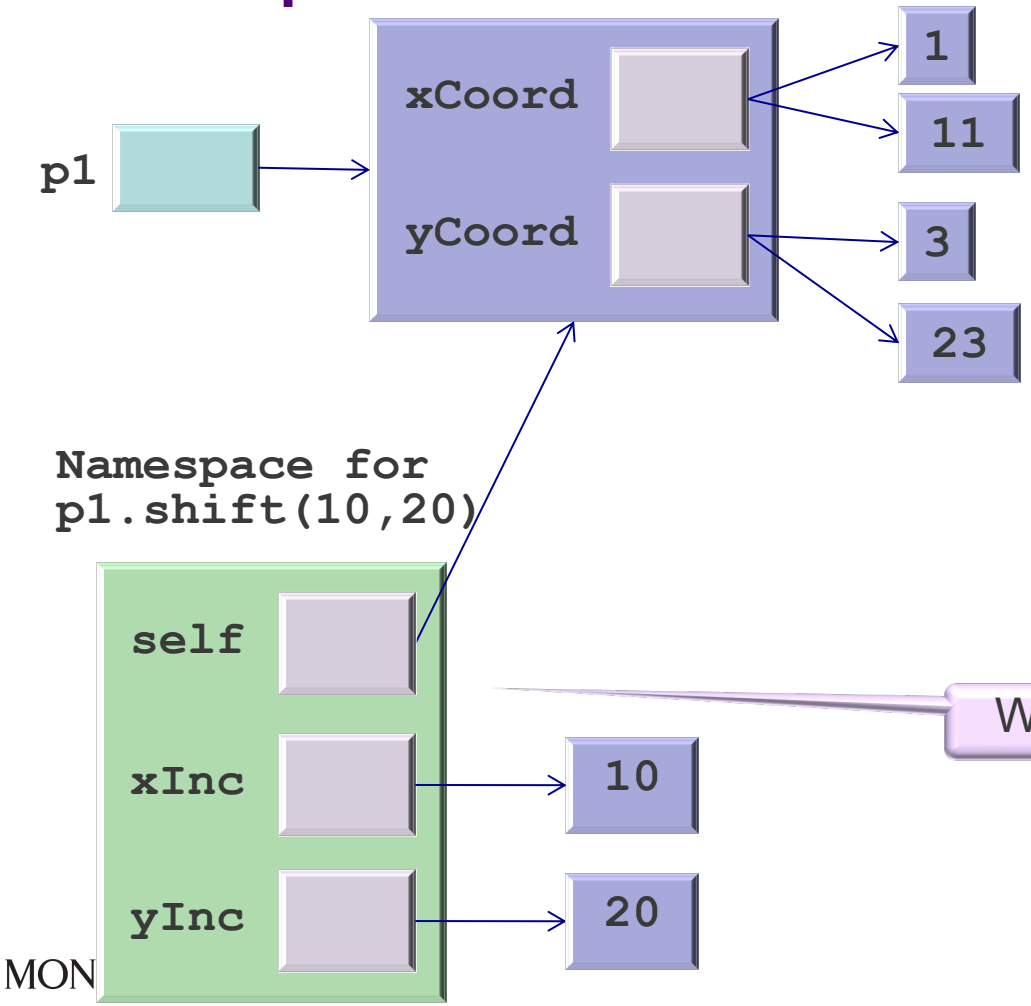
- **A namespace is a mapping of names to objects**
  - Like a dictionary of the variables that can be accessed from that program “block”
- **For example, when the interpreter starts:**
  - It creates a namespace with the names of the built-in functions
- **Each file (also called module) has its own namespace**
  - Don’t put two classes or two functions with the same name in a file
  - They share the same namespace, so the result can be surprising:
    - With two functions, the second definition overwrites the first
- **Functions have their namespace too. When a function is called:**
  - Python creates a local namespace for it
  - That namespace is forgotten once the function ends
- **Names belong to the namespace in which they are bound**

Like the info we created in MIPS

```
x is at 8($fp)
result is at -8($fp)
```

Namespaces (cont)

- **An example with the `Point` class**



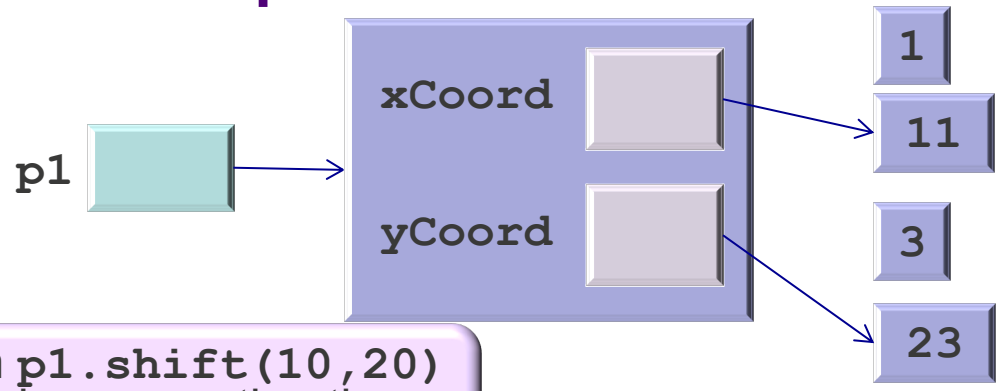

```
class Point:
    def __init__(self, x: float, y: float) -> None:
        self.xCoord = x
        self.yCoord = y

    def shift(self, xInc: float, yInc: float) -> None:
        self.xCoord += xInc
        self.yCoord += yInc
```

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
```

While `p1.shift(10,20)` is executing

Namespaces (cont)

- **An example with the `Point` class**



When `p1.shift(10,20)` finishes executing the namespace is forgotten

But wait! Why were `xInc` and `yInc` in `shift`'s namespace? Were they "bound" by `shift`?

Yes, they were (as parameters); think in MIPS how `shift` would assign the values of 8($fp) and 12($fp) to its parameters

```
class Point:
    def __init__(self, x: float, y: float) -> None:
        self.xCoord = x
        self.yCoord = y

    def shift(self, xInc: float, yInc: float) -> None:
        self.xCoord += xInc
        self.yCoord += yInc
```

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
>>> p1.xCoord
11
>>> p1.yCoord
23
>>>
```

Binding a name

- **There are many ways to bind a name in Python**
- **For example, by:**
  - Assigning to a variable (`x = 13`)
  - Receiving an argument (that's the case for `xInc` and `yInc`)
  - Importing a module (`import x`)
  - Importing anything from a module (`from x import y`)
  - Defining a function (`def x(foo): ...`)
  - Defining a class (`class x: ...`)
  - Writing a for loop (`for x in y: ...`)
  - Writing an except clause (`try: ... except x: ... `)
- **If any of these appears inside a function:**
  - It makes the name local to the function

Scoping

- **Scope: block of text where a namespace is directly accessible**
  - That is, where there is no need to "qualify" the name
    - `from silly import Silly` brings `Silly` to the current namespace
    - Then, there is no need for `silly.Silly`
- **Often there are several scopes in operation, for example:**
  - The scope of the method that is executing
  - The scope of the class where the method is defined
  - The scope of the module where the class is defined
  - The scope of the interpreter that is executing, etc
- **Scope is determined statically but used dynamically**
  - Statically: so that you can always determine the scope of any name by looking at the program
  - Dynamically: because it is at run-time that Python searches for names

Remember qualifying? Related to scoping!

Scoping Rules

- **Remember: names belong to the namespace where they are bound**
  - The scope of a name does not change while the program is running
- **During execution, Python searches for names as follows:**
  - First, in the local scope (which is the innermost)
    - Contains all the names the function or method's namespace
  - Then, in the scope of any enclosing functions/methods (defined inside the other):
    - Searched from the nearest-to-outer enclosing scope
    - Contains nonlocal and nonglobal names (names that are neither local nor global)
  - Then the current module's global names
    - That is, those in the module's namespace
  - Last, the namespace containing built-in names
    - That is, those reserved for Python itself

MONASH University

Exercises for Names, Namespaces, Scopes and Scoping Rules

Example of Scoping Rules

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
7
>>> d()
second
>>> e()
e
```

`a()` has `x` in its local namespace with value `a`

`b()` does not have `x` in its local namespace. And it is not defined within a function. So it looks at the enclosing namespace (module/interpreter) where it finds `x` with value `'first'`

`b()` as before but the value of the global `x` is now `'second'`

`a()` as before

`c()` has `x` in its local namespace with value `7`

`d()` calls `b()` which is as before (the value of the global `x` is still `'second'`)

`e()` defines and calls `f()` which does not have `x` in its local namespace, so it looks in that of its enclosing function and finds it with value `'e'`

How does this relate to “qualifying”?

- **Remember the code used some time before:**

Another way of looking at it: each module is an object of an internal module class. Thus `point.` allows us to access the attributes of object `point` which include the class `Point`

```
class Point:
 def __init__(self, x: float, y: float) -> None:
        self.xCoord = x
        self.yCoord = y
```

Let’s put this code in file `point.py`

We said “why is `point.` needed?”

Because the name `Point` is not directly accessible from the current code, i.e., not in its namespace or in any one where Python will search for it

Qualifying it by `point.` allows us to access the namespace of module `point.` which contains the name `Point`

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.xCoord
1
>>> p1.yCoord
3
>>> p2 = point.Point(-4,7)
>>> p2.xCoord
-4
>>> p2.yCoord
7
>>> p1.__class__
<class 'point.Point'>
```

And how does it relate to this case?

- **Remember the case a few slides before, where:**

```
>>> class Silly
...     i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
>>> Silly.i = 11
```

```
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
>>> Silly.i = 22
>>> s1.i
6
>>> s2.i
22
```

We said: what? if `i` is a class variable, shouldn't it be 6?

No, Because this assignment created a new attribute for `s1` (but not for `s2`) in its local namespace

This will first look in `s1`'s local namespace and will find the newly created attribute

This will first look in `s2`'s local namespace, will find nothing and then look at the class namespace where class variable `i` exists

Scoping rules: A few more examples

- **What would this print?**

```
x = 77

class Mine:
    x = 6
    y = 10
    z = 20
    def __init__(self,x):
        self.y = 12
        self.x = 5
```

```
>>> a = Mine(0)
>>> a
<__main__.Mine object at 0x1006bda10>
>>> a.x
5
>>> a.y
12
>>> a.z
20
>>> b = Mine(0)
>>> a.z = 8
>>> a.h = 9
>>> b.z
20
>>> Mine.x
6
>>> b.h
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Mine' object has no
attribute 'h'
```

Instance variable

Instance variable

Class variable

Creates new instance variables in a

Class variable

Class variable

Non-existent variable in b

Scoping rules: A few more examples

- **What would this print?**

```
x = 67

def one(y):
    if y:
        print(x)
    def x(a):
        return a * 2
    print(x)

def two():
    for x in range(3):
        print(x)

def three():
    class x:
        x = 3
        print(x)
    print(x)
```

`one()` binds `x`, and thus it is local

```
>>> one(False)
<function one.<locals>.x at 0x1006bf440>
>>> one(True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in one
UnboundLocalError: local variable 'x'
referenced before assignment
>>> two()
0
1
2
>>> three()
3
<class '__main__.three.<locals>.x'>
```

Local variable that is not yet bound

local variable

Scoping rules: A few more examples

- **What would the following print?**

```
x = [4]

def a():
    x.append(3)

def b():
    x = [1, 2]
    x.append(3)
    print(x)

def c(x):
    for x in range(3):
        print(x)

def d():
    print(x)
    x = 7
    print(x)
```

```
>>> a()
>>> x
[4,3]
>>> b()
[1,2,3]
>>> c(100)
0
1
2
>>> d()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in d
UnboundLocalError: local variable 'x'
referenced before assignment
```

Global variable

Local variable

Local variable

Local variable not yet bound

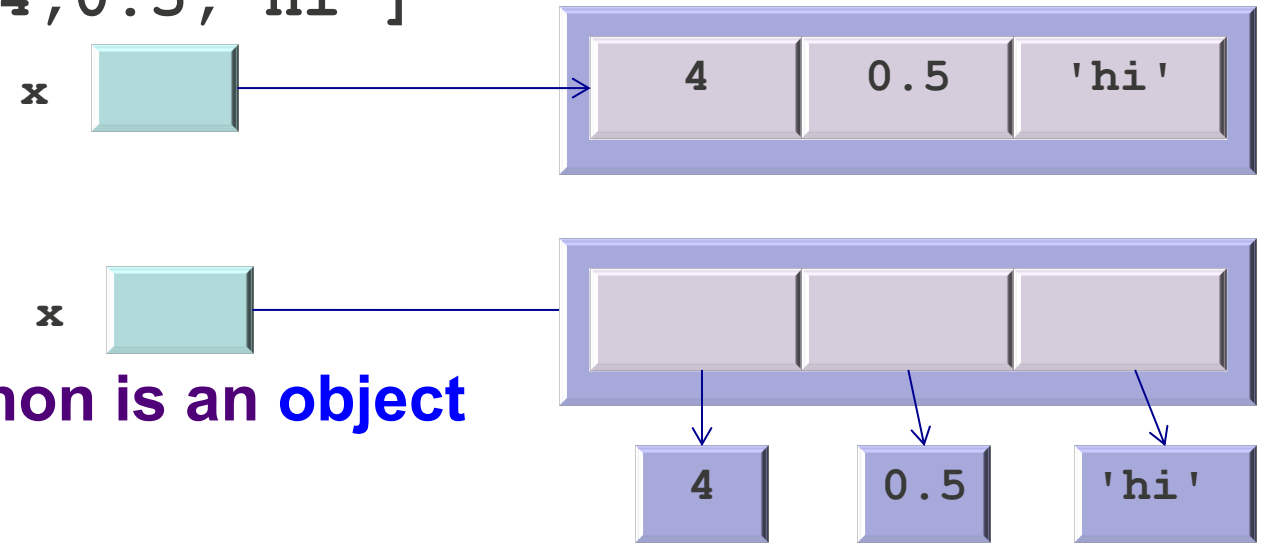
MONASH University

Mutable and Immutable Objects

Let's first look at Python lists

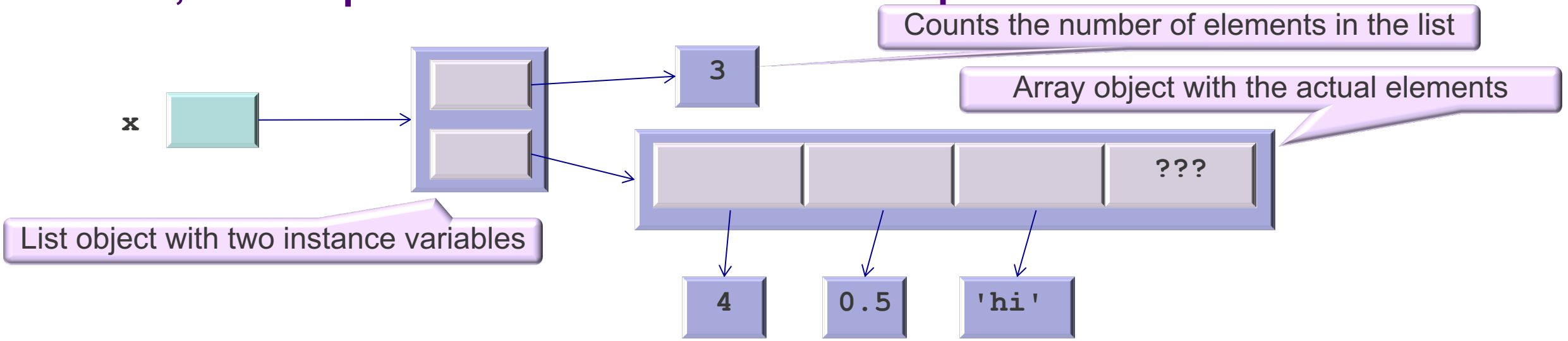
- **How are Python lists represented internally?**
  - They are implemented as arrays
  - But they are also objects
    - Like any object, they will have type, value and stuff…
    - And we will visualise it as before:

  - Question is: how is the value of a Python list represented?
- **Consider the list** `x = [4,0.5,'hi']`
- **Like this?**
- **Close, but not quite**
- **More like this:**
- **Since every value in Python is an object**



Let's first look at Python lists (cont)

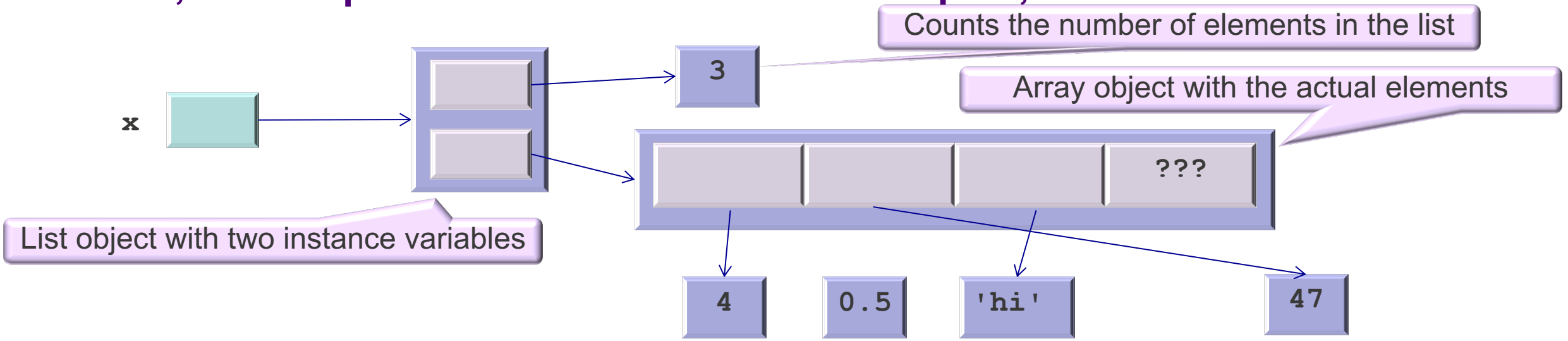
- **In fact, their implementation is a bit more complex:**



- **The 3 says that only the first 3 cells in the array (capacity 4) are used**
- **The important point is that they are arrays of references**
- **What does `x[1] = 47` do?**

Let's first look at Python lists (cont)

- **In fact, their implementation is a bit more complex, as we will see later:**



- **The 3 says that only the first 3 cells in the array (capacity 4) are used**
- **The important point is that they are arrays of references**
- **What does `x[1] = 47` do?**
  - Simply changes the reference, nothing else!
  - This modifies the object referred to (down the line) by **`x`**, but not **`x`**

Assigning variables versus list elements

- **Let's compare the two pieces of code:**

```
>>> x = 10
>>> y = x
>>> x = 'hi'
>>> x;y
'hi'
10
```

```
>>> x = [4,0.5,'hi']
>>> y = x
>>> x[1] = 47
>>> x;y
[4,47,'hi']
[4,47,'hi']
```

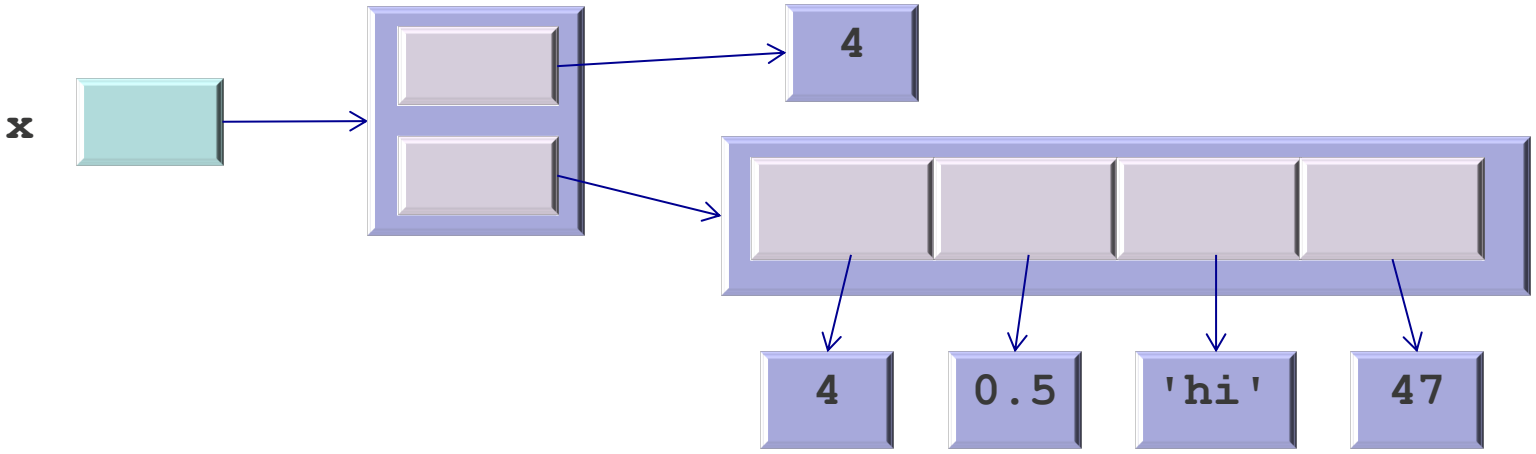
I was re-assigning an element in the list, not x

- **They behave differently!**
- **Not really. If I do:**
- **It prints:**
- **So the same as before**

```
>>> x = [4,0.5,'hi']
>>> y = x
>>> x = [9,-1]
>>> x;y
[9,-1]
[4,0.5,'hi']
```

Python lists are mutable

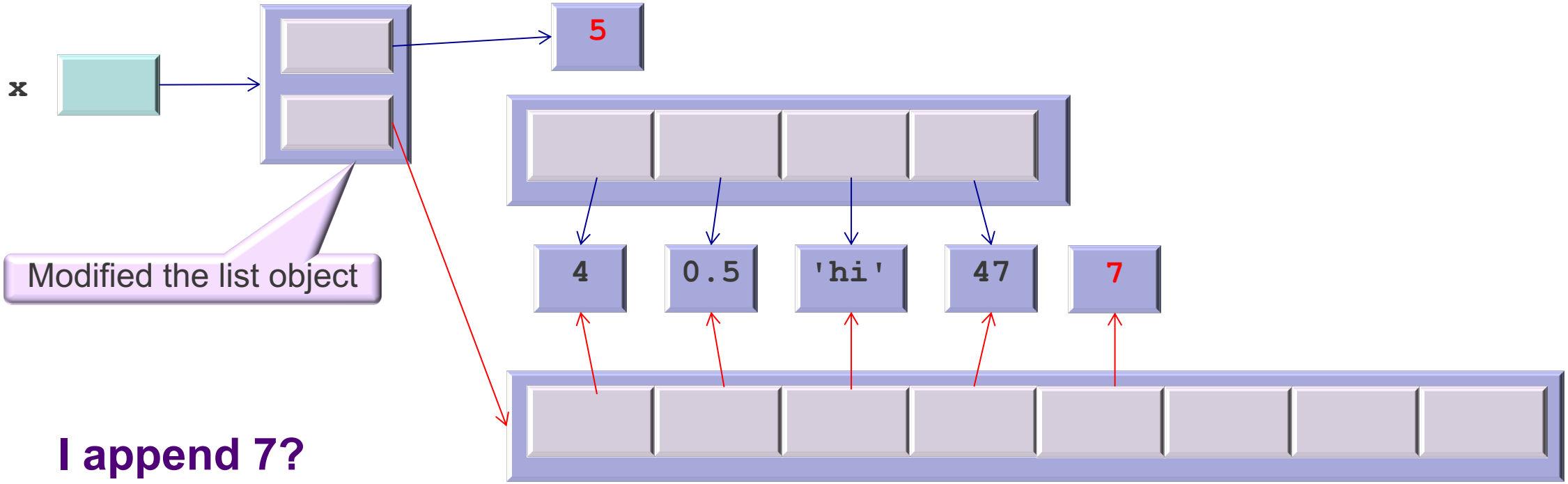
- What happens if given the list:



I append 7?

Python lists are mutable

- **What happens if given the list:**



I append 7?

- **The list object itself is modified. Which means Python list are mutable:**
  - In other words: objects of type list in Python can be changed

Can I modify an integer object then?

Numbers, strings and tuples are immutable. Why? We will see later in the unit.

- **No, they are immutable, I can only create integer objects:**
  - Once created they cannot be changed
  - I can create a new one, but not modify an already created one
- **Is this because they are "atomic" (indivisible)?**
  - No: tuples are not atomic and they are also immutable
- **Really?**

Let's see:

```
>>> y = (0.5, 7, 31)
>>> y
(0.5, 7, 31)
>>> y[0]
0.5
>>> y[0] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

This is how you access element 0 in a tuple

Exception since tuples cannot be modified: are immutable

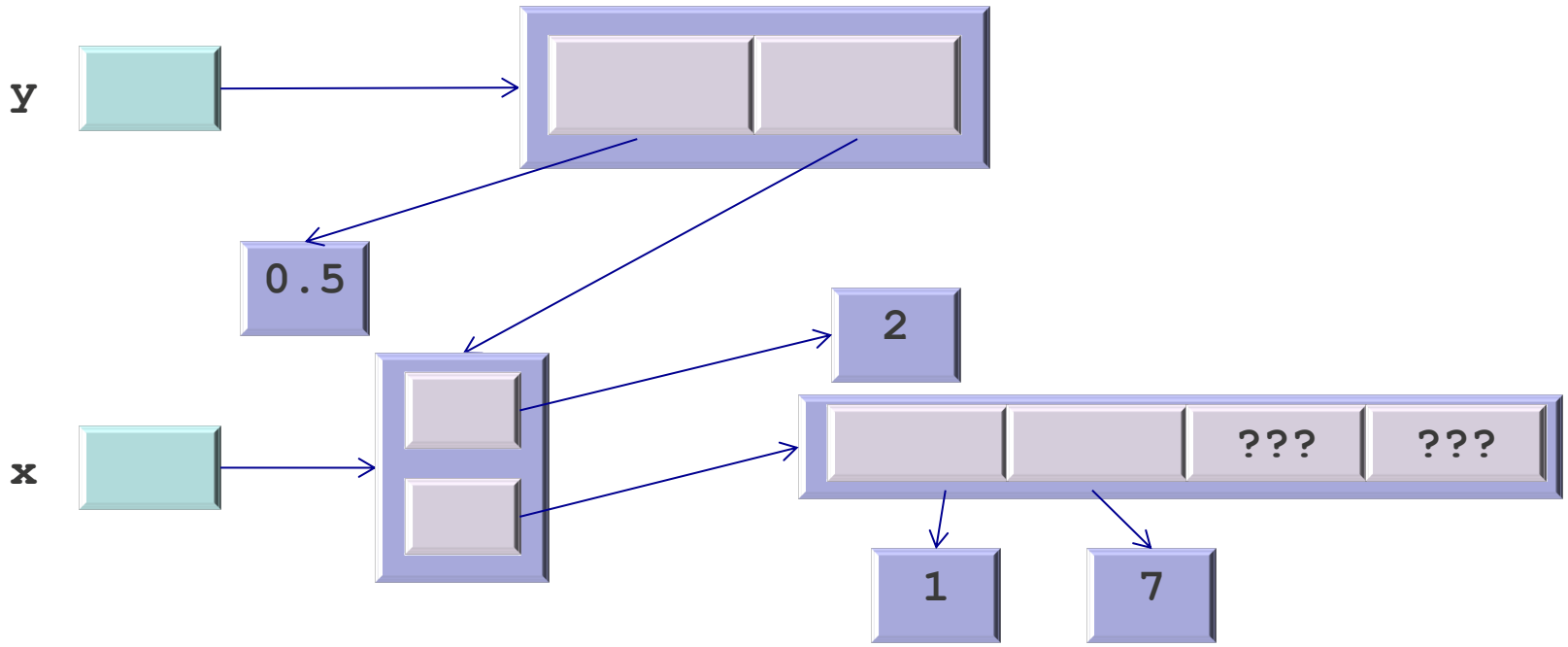
```
>>> x = 3
>>> y = (4,x)
>>> y
(4,3)
>>> x = 5
>>> y
(4,3)
>>>
```

Still the same!

Yes, tuples really are immutable

- **You might think you have modified tuples before**
- **Let's see:**

```
>>> x = [1,7]
>>> y = (0.5,x)
>>> x[0] = 10
```

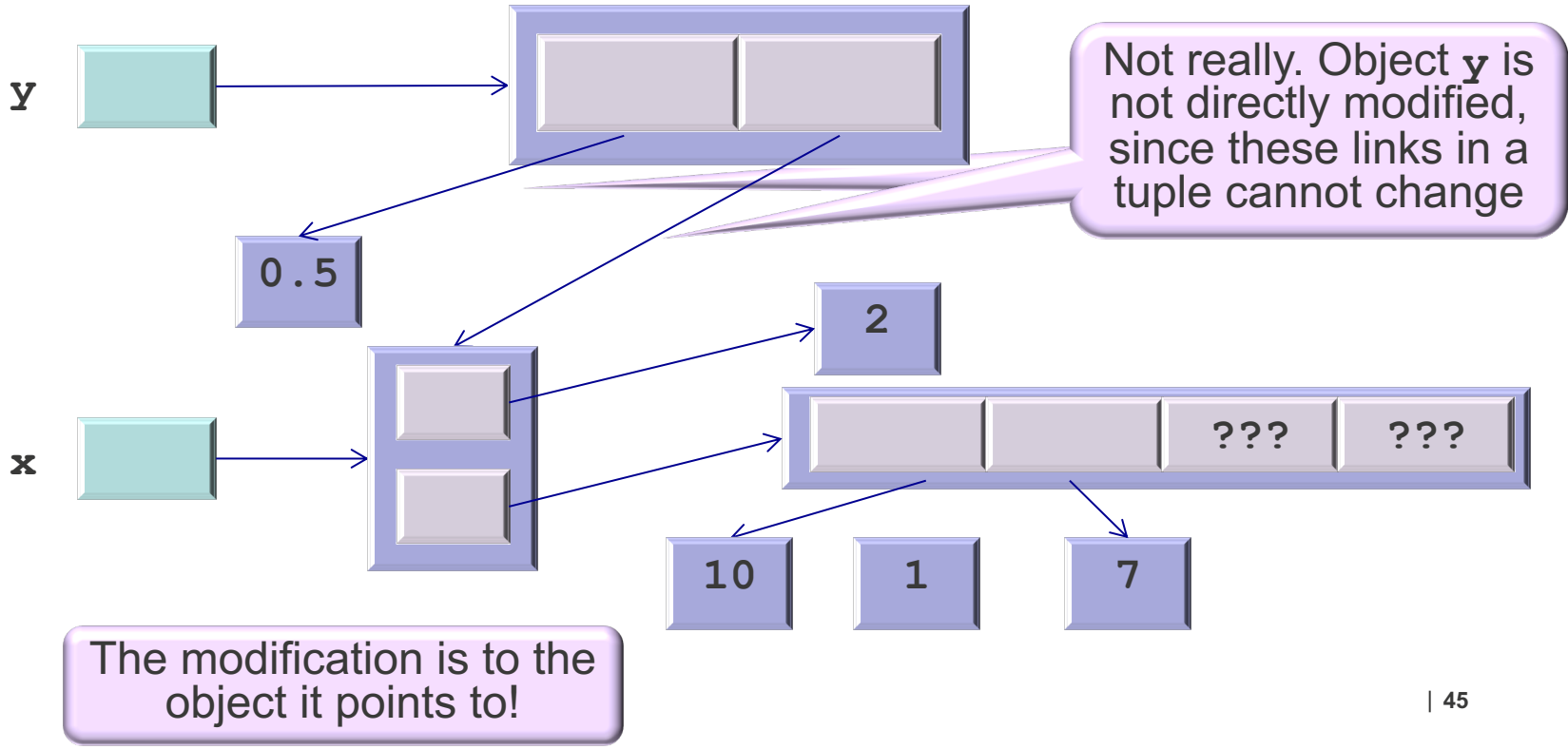


Yes, tuples really are immutable

- **You might think you have modified tuples before**
- **Let's see:**

```
>>> x = [1,7]
>>> y = (0.5,x)
>>> x[0] = 10
>>> x;y
[10, 7]
(0.5, [10, 7])
```

Modified…



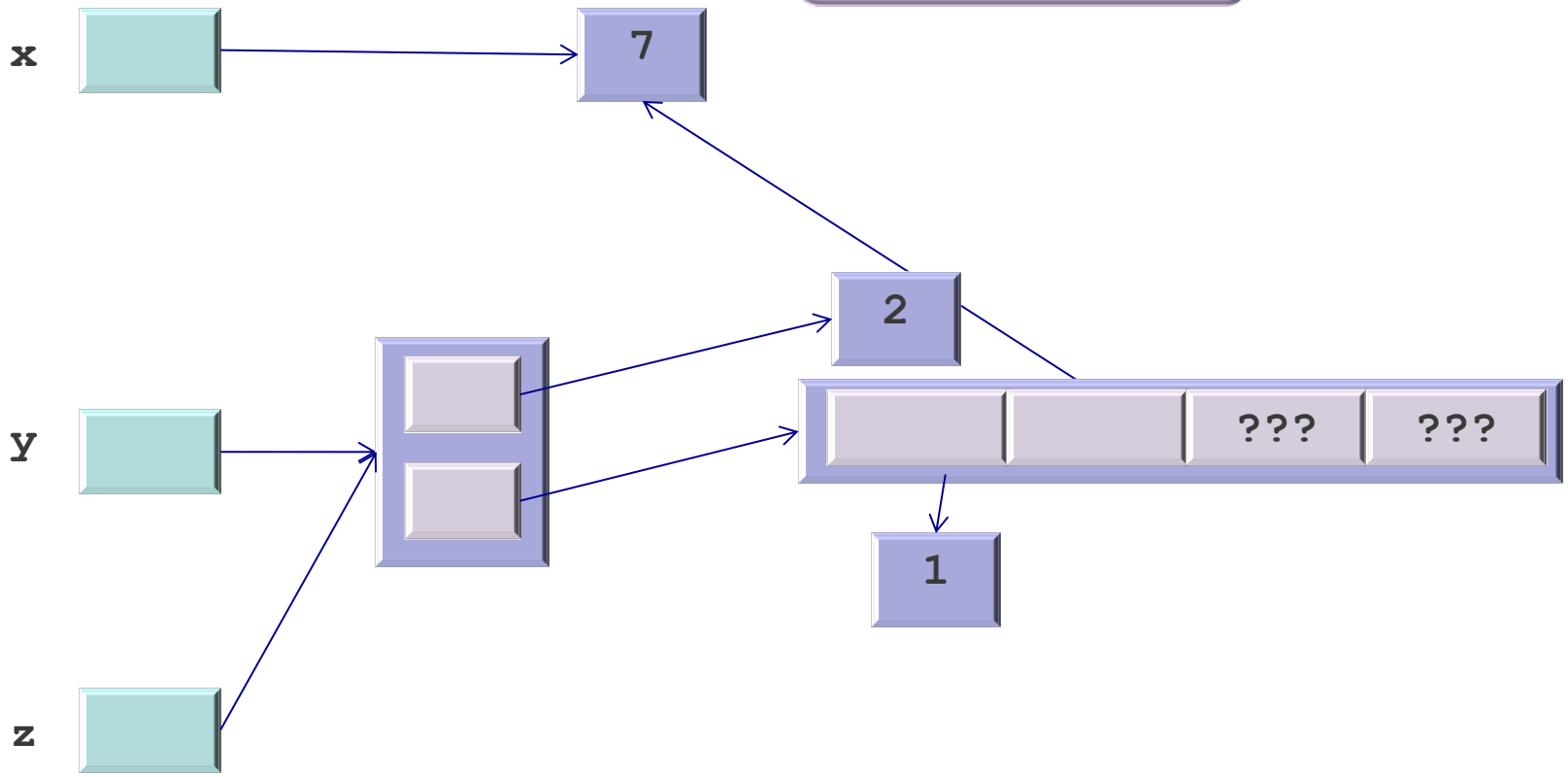
Your turn

- **What does the following code print?**

```
>>> x = 7
>>> y = [1,x]
>>> z = y
>>> z[1] = 9
>>> x;y;z
```

After the first 3 lines it looks like this

And after the 4th line, it looks like?



Summary

- **We know how OO in Python affects:**
  - Variable and value creation and assignment in Python
  - Names and Namespaces,
  - Scopes and Scoping Rules
- **We are able to follow Python code involving**
  - Variable assignments and aliasing
  - Mutable and immutable objects