CISS450: Artificial Intelligence Lecture 3: Execution Environment & Memory Model

Yihsiang Liow

Agenda

- Python byte codes
- Python Virtual Machine
- Alternative Execution Models
- Name and Object Space Model

Python Byte Code and PVM

- Python is an interpreted language
- Your source code is compiled into byte codes
- Byte code is platform-independent
- Byte code is executed through a Python Virtual Machine (PVM)
- PVM read byte code instructions and execute them one at a time

Compiler

- A C/C++ compiler outputs of assembly language code which is then translated into machine code by an assembler. The machine code is executed directly by the CPU.
- Python byte code runs slower than the executable generated from C/C++ source code.
- Python's execution model is similar to that of Java, C#

Saved Byte Code

- If you run a standalone Python source code, the byte code is not saved
- If you import Python code, then the byte code generated from the imported file is saved. These files have `.pyc` extensions.
- The saved `.pyc` files reduces repeated byte code generation
- We will talk about importing modules later

Alternatives

- It is possible to generate executable machine code from Python code. The executable is not platform-independent.
- Jython generates Java byte code
- Python .NET (IronPython) generates CIL byte codes
- Psyco Just-in-Time compiler: As byte code is executed, Psyco will generate machine code to replace byte code. Execution speed increases with time. Superseded by PyPy.
- Stackless Python; See EVE Online.

Name and Object Space Model

- Now for something different
- Let's talk about how Python treats names (example: variables) and their values (i.e., associated object)
- You already know that in Python you do not declare types for variables. Example:

```
x = 100
```

Furthermore, the type of a variable can change:

```
x = 100
x = "One hundred"
```

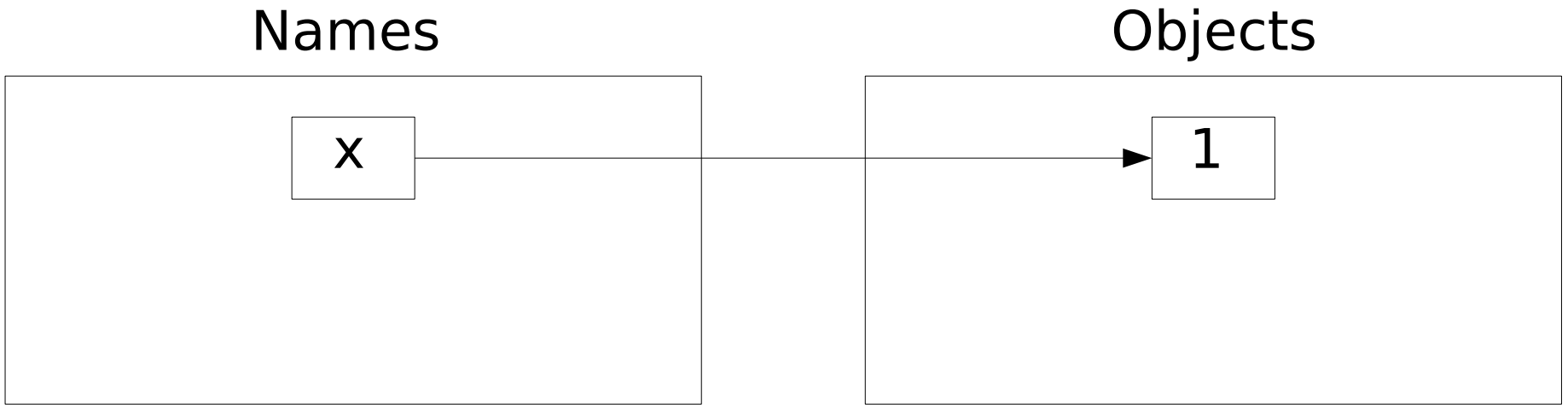
Name and Object Space Model

- This language feature is known as ***dynamic typing***. The type of a variable is known (and might change) during runtime.
- In comparison, C/C++ is a ***static typing*** language.
- What happens when you execute the following?

```
x = 1
```

the following is a model of Python's runtime memory.

Name and Object Space Model



- For those with C/C++ background, you can think of x being a pointer but you need not de-reference x. Or you can think of x as a reference.
- There are two types of objects: mutable and immutable.

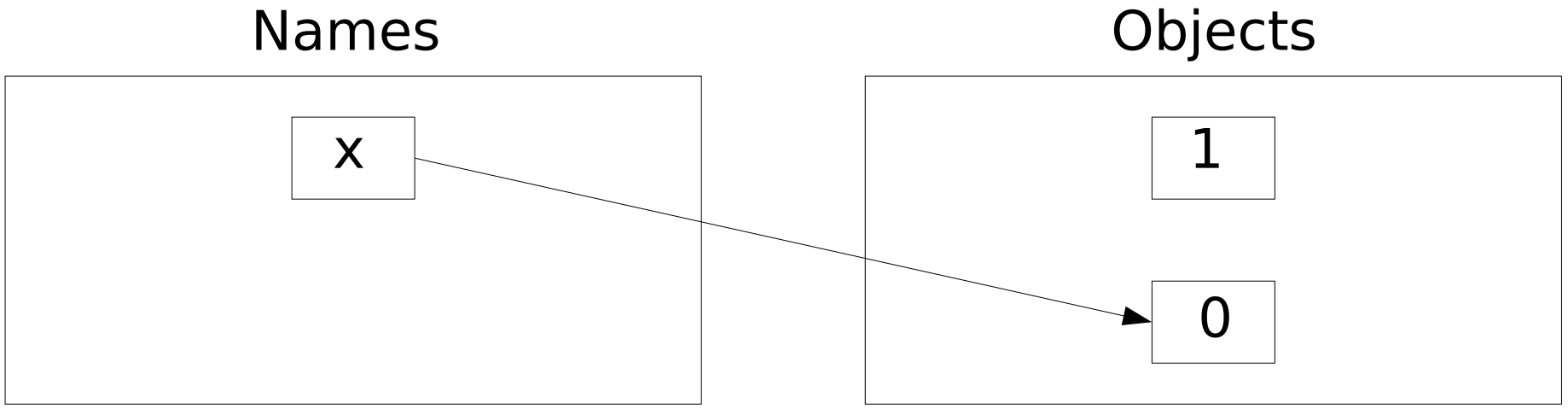
Names and Object Space Model

- Mutable: You can change the object's state (values)
- Immutable: Not mutable
- Integers are immutable. So, you ***cannot*** change the value 1 in the above picture to 0.
- But ... in that case how do we change the value of x??? For instance what if we want:

```
x = 1
x = 0
```

- This is what happens ...

Name and Object Space Model



- The object 1 is not referenced (not used)
- But what is the problem? You should know this from CISS240. What is that big scary thing that can happen with pointers?

Reference Counting and gc

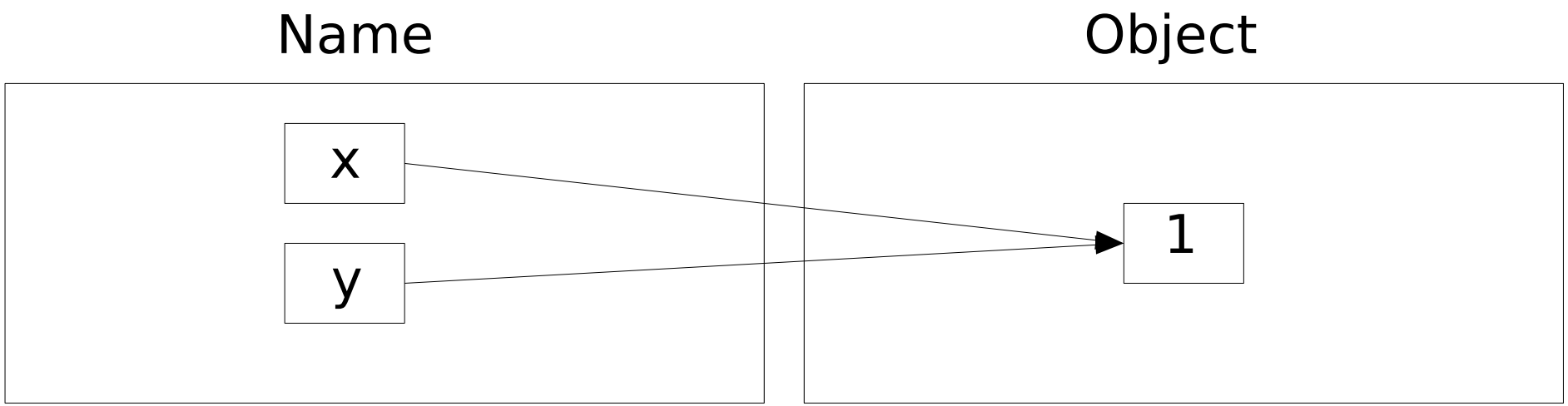
- Each object has a ***reference count***
- The reference count is the number of names/objects referencing the object. When the reference count is zero, the object is not being used.
- Python automatically scans objects and prepares those with zero reference count for deletion, i.e., removing them from memory. This is known as ***garbage collection***.
- Garbage collector can be tweaked

Object Reuse

- For optimization, Python reuses some objects
- So for instance when you have:

```
x = 1
y = 1
```

the object 1 is the ***same*** object

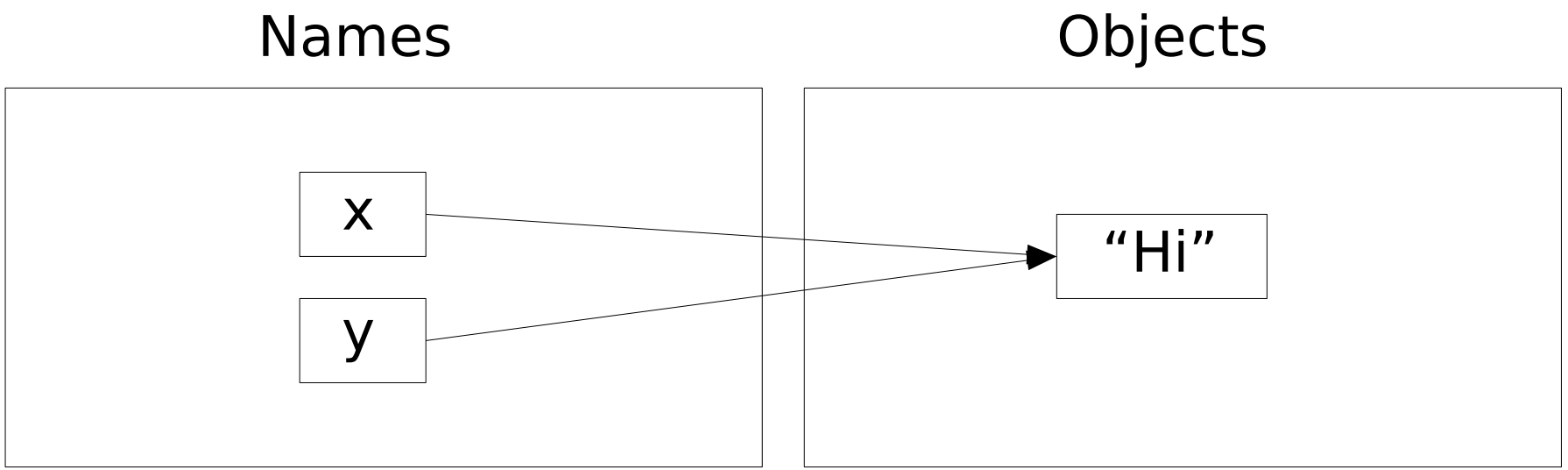


Object Reuse

- Short strings are also reused. Example:

```
x = "Hi"
y = "Hi"
```

Then x and y references the same `"Hi"` string object.



Warning on Assignment

- WARNING: Beware of assignments! When you do

```
x = y
```

x and y refers to the ***same*** object. Modifying the object through x will modify the object y is referencing if the object they are referring to is mutable. Integers, floats, strings are immutable so that's not a problem. Later we'll see mutable types.

Object Identity

- Object identity: Python gives each object a unique object id. For the standard Python downloaded from www.python.org, the id is the location in memory. If x refers to 1 and you want the id of integer object with value 1, use id(x):

```
>>> x = 1
>>> y = 1
>>> print(id(x), id(y))
7678960 7678960
```

- id is a built-in function

Type

- You have already seen `type`. If you have

```
x = 1
```

  then `type(x)` refers to the type of the object that x references

- type(x) returns a type object. You can compare type objects. The predefined types are in types module.

```
import types
x = 1
if type(x)==int:
    print("x is an integer")
```

Type

- You can also do this (which is better):

```
x = 1
if isinstance(x, int):
    print("x is an integer")
```

- The function call `isinstance(x, C)` return `True` iff `x` is an object created from the class `C`.
- Try these:

```
print(isinstance("hello world", str))
print(isinstance(1.2, float))
print(isinstance("hello world", int))
print(isinstance("hello world",(int,str)))
```

Import

- Here is a short note on import. We will cover more later.
- Since Python is slower, sometimes it's important to improve on your implementation.
- To find out which implementation of two code snippets is faster, it's convenient to use the `time` module.
- A module is just a python program file.
- There are several ways of importing a module.

Import

- Here is one:

WARNING: NOT **time.py**!

clock() is a function in the **time** module

```
import time
start = time.clock()
x = 1
for i in range(1, 10001):
      x = x * i
end = time.clock()
print("time taken:", end - start)
```

Import

- (Also, see `resources` module.)
- To check what variables/functions/classes/etc are available in the module, check python documentation or use the web.
- You can also see what are available in the module like this:

```
import time
dir(time)
print(time.clock.__doc__)
```