Python’s dynamic typing model
+
a quiz as a reminder

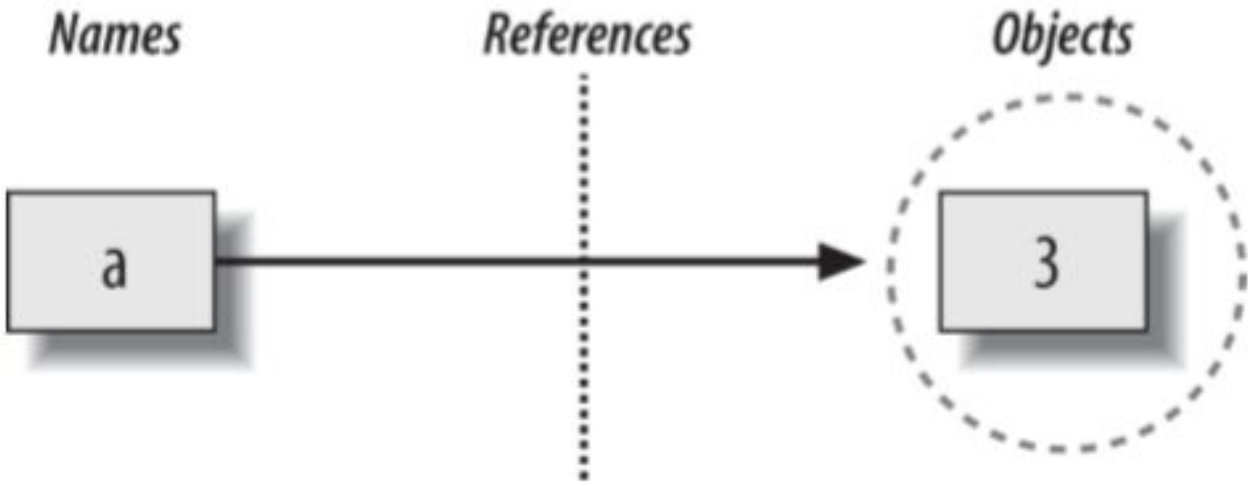
Types are determined automatically at runtime, not in response to declarations in your code

Variable creation: A variable (i.e., name), like ***a***, is created when your code first assigns it a value. Future assignments change the value of the already created name. Technically, Python detects some names before your code runs, but you can think of it as though initial assignments make variables.
Variable types: A variable never has any type information or constraints associated with it. The notion of type lives with objects, not names. Variables are generic in nature; they always simply refer to a particular object at a particular point in time.

Variable use: When a variable appears in an expression, it is immediately replaced with the object that it currently refers to, whatever that may be. All variables must be explicitly assigned before they can be used; referencing unassigned variables results in errors.

a = 3

- Create an object to represent the value 3.
- Create the variable a, if it does not yet exist.
- Link the variable a to the new object 3.



What is what?

- ***Variables*** are entries in a system table, with spaces for links to objects.
- ***Objects*** are pieces of allocated memory, with enough space to represent the values for which they stand.
- ***References*** are automatically followed pointers from variables to objects.

(Python internally caches and reuses certain kinds of unchangeable objects)

Types Live with Objects, Not Variables

- Objects have more structure than just enough space to represent their values.
- Each object also has two standard header fields:

a ***type designator*** used to mark the type of the object (for example, int, str, etc.);

a ***reference counter*** used to determine when it's OK to reclaim the object.

How does it work?

```
>>> a = 3
>>> a = 'spam'
>>> a = 1.23
```

Reminder: each object contains a header field that tags the object with its type. The integer object 3 contains the value 3, plus a designator - a pointer to an object called *int*. The type designator of the 'spam' string object points to the string type (called *str*). Because objects know their types, variables don’t have to.

Objects Are Garbage-Collected

>>> a = 3

>>> a = 'spam‘

What happens to object 3?

Each time *a* is assigned to a new object, Python reclaims the prior object’s space. References to objects are discarded along the way. As soon as (and exactly when) a counter that keeps track of the number of references drops to zero, the object’s memory space is automatically reclaimed.

Cyclic references

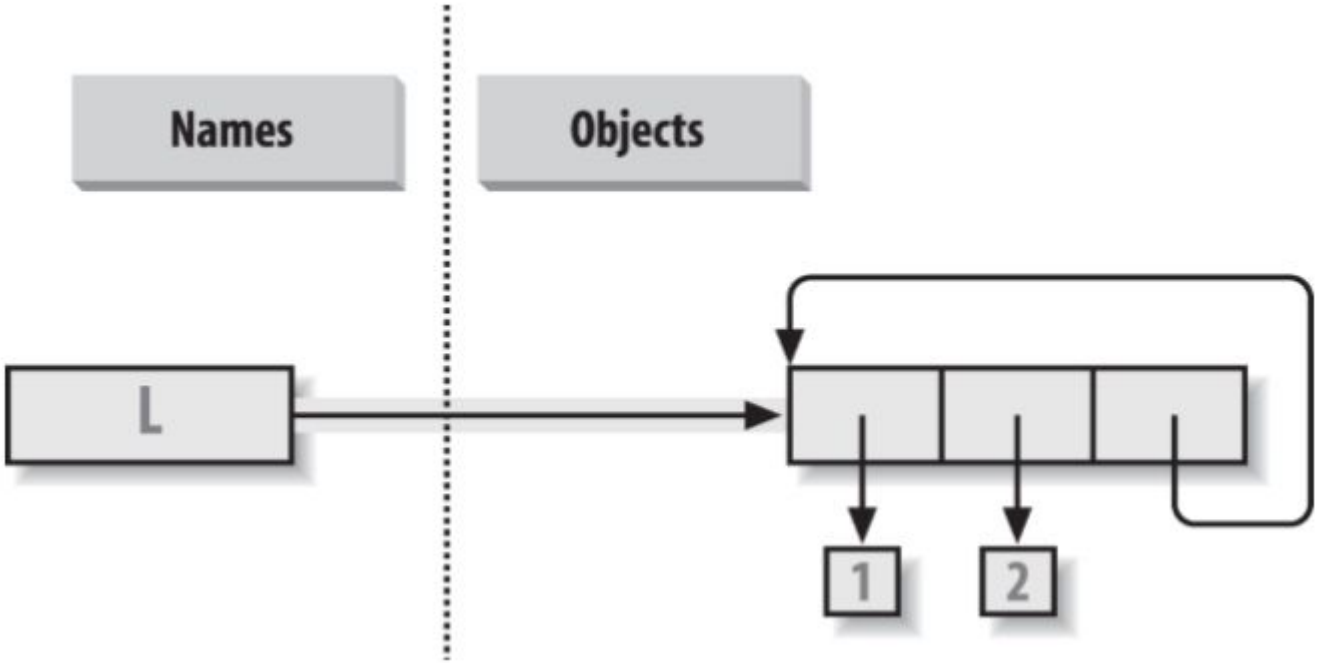
Because references are implemented as pointers, it's possible for an object to reference itself, or reference another object that does.

Try it:

```
>>> L = [1, 2]
>>> L.append(L)
```

What happened? Why?

Cyclic references



Old style: [1, 2, [1, 2, [1, 2, [1, 2, and so on

New style: [1, 2, [...]]

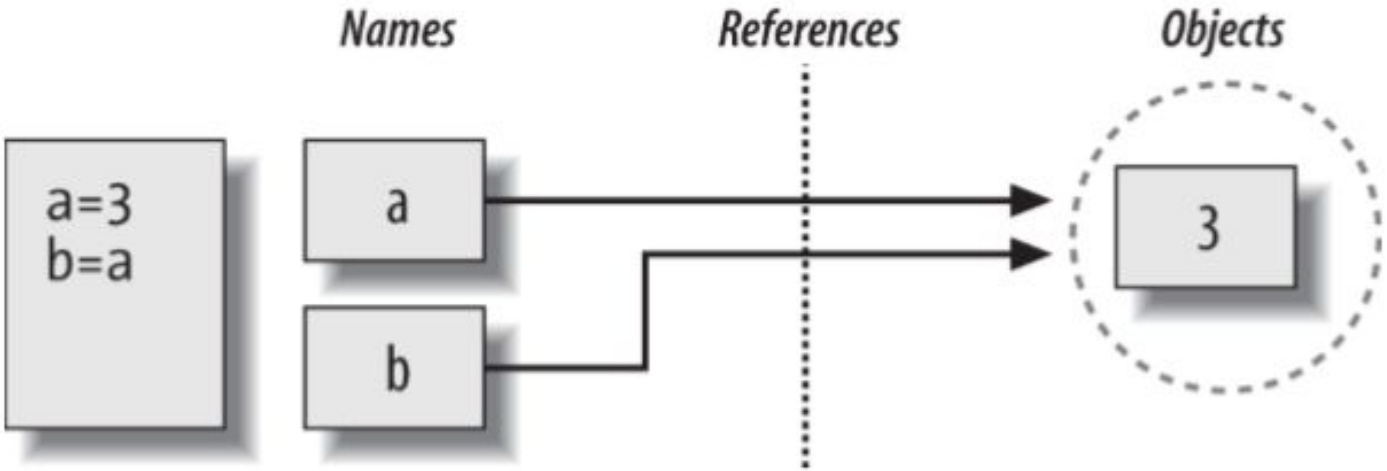
Cyclic references

Assignments in Python always generate references to objects, not copies of them. You can think of objects as chunks of memory and of references as implicitly followed pointers. When you run the first assignment, the name L becomes a named reference to a two-item list object—a pointer to a piece of memory. Python lists are really arrays of object references, with an append method that changes the array in place by tacking on another object reference at the end. Here, the append call adds a reference to the front of L at the end of L, which leads to the cycle.

Because the reference counts for such objects never drop to zero, they must be treated specially (see **gc** module in Python's library manual).

Shared References

```
>>> a = 3
>>> b = a
```

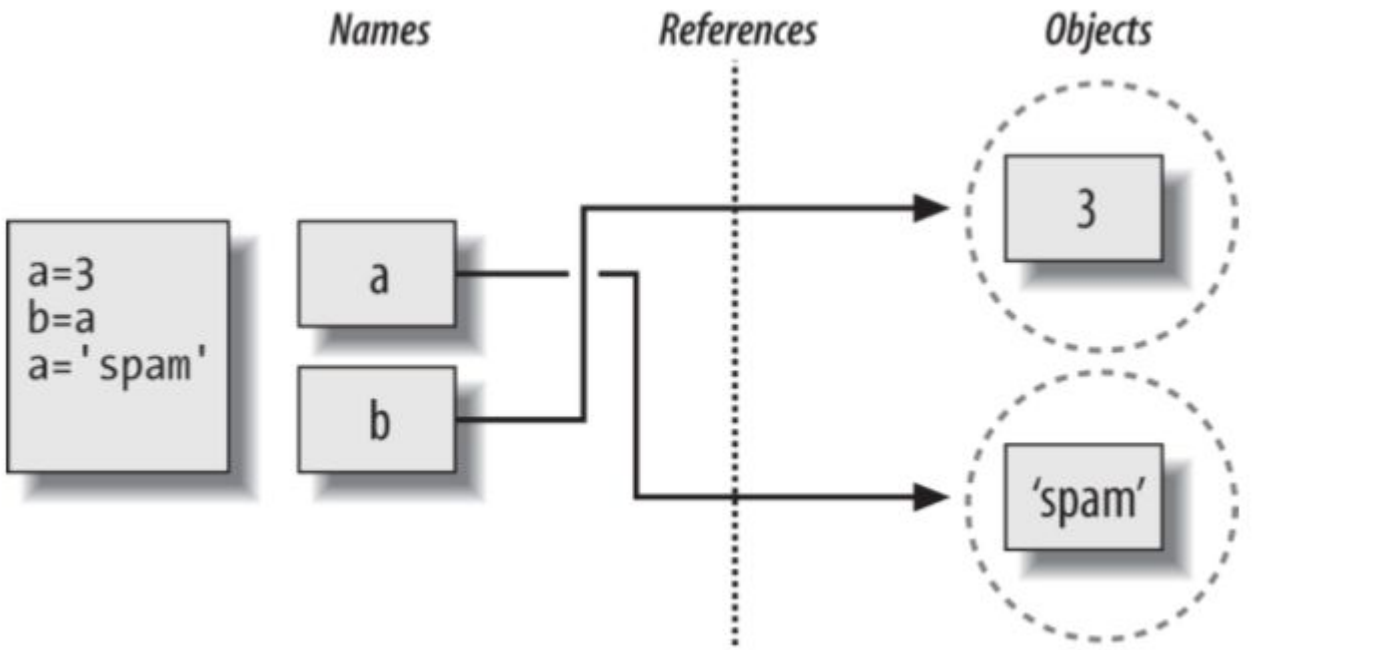


names *a* and *b* are not linked to each other directly!!!

there is no way to ever link a variable to another variable in Python !!!

Shared References

```
>>> a = 3
>>> b = a
 >>> a = 'spam‘
```



Shared References

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

What happens?

(the last assignment sets ***a*** to a completely different object -in this case, the integer 5, which is the result of the + expression. It does not change ***b*** as a side effect. In fact, there is no way to ever overwrite the value of the object 3—as integers are immutable and thus can never be changed in place.

Shared References and In-Place Changes

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
>>> L1 = 24
```

What is the difference?

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
>>> L1[0] = 24
```

Avoiding side effects

Side effect: L1=[24, 3, 4] and L2=[24, 3, 4]

```
>>> L1 = [2, 3, 4]
 >>> L2 = L1[:]      (or list(L1), copy.copy(L1), etc.)
>>> L1[0] = 24
```

Result:

L1 = [24, 3, 4]

L2 is not changed [2, 3, 4]

to copy a dictionary or set, use their X.copy() method call or their type names, **dict** and **set**

Making copies

Standard library **copy** module has a call for copying any object type generically, as well as a call for copying nested object structures—a dictionary with nested lists, for example:

```
import copy
X = copy.copy(Y)          # Make top-level "shallow" copy of any object Y
X = copy.deepcopy(Y)      # Make deep copy of any object Y: copy all nested parts
```

Shared References and Equality

```
>>> L = [1, 2, 3]
>>> M = L                  # M and L reference the same object
>>> L == M                 # Same values
True
>>> L is M                 # Same objects
True
```

== operator, tests whether the two referenced objects have ***the same values***;

is operator, instead tests for object identity—it returns **True** only if both names point to the exact same object, so it is a much stronger form of equality testing.

Why is so?

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]        # M and L reference different objects
>>> L == M
True
>>> L is M                # Different objects
False

>>> X = 42
>>> Y = 42                # Should be two different objects
>>> X == Y
True
>>> X is Y                # Same object anyhow: caching at work!
 True
```

(small integers and strings are cached and reused not literally reclaimed; they will likely remain in a system table to be reused the next time you generate a them in your code)

Summary

Dynamic Typing Is Everywhere

Embedded types (reminder)
The basics

```
"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)
'green {0} and {1}'.format('eggs', S)
('x',)[0]
('x', 'y')[1]
L = [1,2,3] + [4,5,6]
L, L[:], L[:0], L[-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
```

```
L.reverse(); L
L.sort(); L
L.index(4)
{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)] = 4
list(D.keys()), list(D.values()), (1,2,3) in D
[[]], ["",[],(),{},None]
```

Embedded types (reminder)
Indexing and slicing

L=[0,1,2,3]

a. What happens when you try to index out of bounds (e.g., L[4])?
b. b. What about slicing out of bounds (e.g., L[−1000:100])?
c. Finally, how does Python handle it if you try to extract a sequence in reverse, with the lower bound greater than the higher bound (e.g., L[3:1])? Try L[3:1:-1].

Hint: try assigning to this slice (L[3:1]=['?']), and see where the value is put. Do you think this may be the same phenomenon you saw when slicing out of bounds?

Embedded types (reminder) Indexing, slicing, and del

Define some list L with four items, and assign an empty list to one of its offsets (e.g., L[2]=[]). What happens?

Then, assign an empty list to a slice (L[2:3]=[]). What happens now?

Recall that slice assignment deletes the slice and inserts the new value where it used to be.

The del statement deletes offsets, keys, attributes, and names. Use it on your list to delete an item (e.g., del L[0]). What happens if you delete an entire slice (del L[1:])? What happens when you assign a nonsequence to a slice (L[1:2]=1)?

Embedded types (reminder)
Tuple assignment

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
```

What do you think is happening to X and Y?

The values of X and Y are swapped. When tuples appear on the left and right of an assignment symbol (=), Python assigns objects on the right to targets on the left according to their positions. This is probably easiest to understand by noting that the targets on the left aren't a real tuple, even though they look like one; they are simply a set of independent assignment targets. The items on the right are a tuple, which gets unpacked during the assignment (the tuple provides the temporary assignment needed to achieve the swap effect).

Embedded types (reminder)
Dictionary keys

Consider the following code fragments:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

You've learned that dictionaries aren't accessed by offsets, so what's going on here? Does the following shed any light on the subject? (Hint: strings, integers, and tuples share which type category?)

```
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

Embedded types (reminder)
Dictionary indexing

Create a dictionary named D with three entries, for keys 'a', 'b', and 'c'.

What happens if you try to index a nonexistent key (D['d'])?

What does Python do if you try to assign to a nonexistent key 'd' (e.g., D['d']='spam')?

How does this compare to out-of-bounds assignments and references for lists?

Does this sound like the rule for variable names?

Indexing a nonexistent key (D['d']) raises an error; assigning to a nonexistent key (D['d']='spam') creates a new dictionary entry. On the other hand, out-of-bounds indexing for lists raises an error too, but so do out-of-bounds assignments. Variable names work like dictionary keys; they must have already been assigned when referenced, but they are created when first assigned.

Embedded types (reminder)
Generic operations

Run interactive tests to answer the following questions:

a. What happens when you try to use the + operator on different/mixed types (e.g., string + list, list + tuple)?
b. Does + work when one of the operands is a dictionary?
c. Does the append method work for both lists and strings? How about using the keys method on lists? (Hint: what does append assume about its subject object?)
d. Finally, what type of object do you get back when you slice or concatenate two lists or two strings?

Embedded types (reminder)
String indexing

Define a string S of four characters: S = "spam". Then type the following expression: S[0][0][0][0][0]. Any clue as to what's happening this time? (Hint: recall that a string is a collection of characters, but Python characters are one-character strings.) Does this indexing expression still work if you apply it to a list such as ['s', 'p', 'a', 'm']? Why?

Every time you index a string, you get back a string that can be indexed again. This generally doesn't work for lists (lists can hold arbitrary objects) unless the list contains strings.

Embedded types (reminder)
Immutable types

Define a string S of four characters again: S = "spam". Write an assignment that changes the string to "slam", using only slicing and concatenation. Could you perform the same operation using just indexing and concatenation? How about index assignment?

Embedded types (reminder)
Nesting

Write a data structure that represents your personal information: name (first, middle, last), age, job, address, email address, and phone number. You may build the data structure with any combination of built-in object types you like (lists, tuples, dictionaries, strings, numbers). Then, access the individual components of your data structures by indexing. Do some structures make more sense than others for this object?

Embedded types (reminder)
Files

Write a script that creates a new output file called myfile.txt and writes the string "Hello file world!" into it.

Then write another script that opens myfile.txt and reads and prints its contents.

Problems to solve

- The game “Find words”. One should find words which are present on the game field of symbols.
- There given a set of cities’ names. One should find all the sequences that meet the rules of the well known game.
- There given the data of students’ exams results in the form of the sheets on each subject. Each sheet has the list of students with their scores in this very subject. One should get the list of failed students in the following form: the name of the student and the list of subjects he\she failed (score less than 50).