Chapter 5- The Dynamic Typing

The Case of the Missing Declaration Statements

- In Python, types are determined automatically at runtime, not in response to declarations in your code.
- you never declare variables ahead of time

Variables, Objects, and References

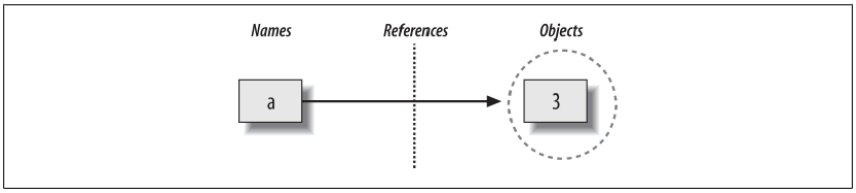
- *Variable creation:*
  - is created when your code first assigns it a value.
  - Future assignments change the value of the already created name.
  - Python detects some names before your code runs, but you can think of it as though initial assignments make variables.
- *Variable types:*
  - A variable 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.
  - all variables must be explicitly assigned before they can be used; referencing unassigned variables results in errors.

e.g:

```
a = 3     # Assign a name to an object
```

Python will perform 3 steps:

- 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`.



- variables and objects are stored in different parts of memory and are associated by links.
- Variables always link to objects and never to other variables
- larger objects may link to other objects (for instance, a list object has links to the objects it contains).

- These links from variables to objects are called ***references*** in Python
- a reference is a kind of association, implemented as a pointer in memory.
- Whenever the variables are later used (i.e., referenced), Python automatically follows the variable-to-object links.

In concrete terms:

- *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.

- As an optimization, Python internally caches and reuses certain kinds of unchangeable objects, such as small integers and strings.
- from a logical perspective, it works as though each expression's result value is a distinct object and each object is a distinct piece of memory.

- Each object also has two standard header fields: a ***type designator*** used to mark the type of the object, and a ***reference counter*** used to determine when it's OK to reclaim the object.

Types Live with Objects, Not Variables

```
a = 3     # It's an integer
a = 'spam'       # Now it's a string
a = 1.23         # Now it's a floating point
```

- ***Names*** have no types
- types live with objects
- in above example, we've simply changed '**a'** to reference different objects
- Because variables have no type, we haven't changed the type of the variable '**a'**; we've simply made the variable reference a different type of object.

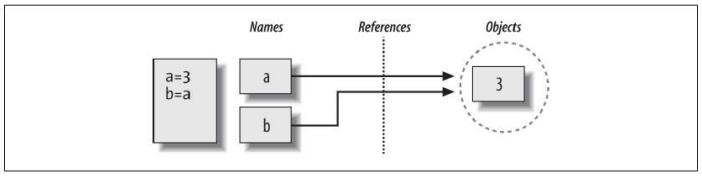
- ***Objects***, on the other hand, know what type they are—each object contains a header field that tags the object with its type.

Objects Are Garbage-Collected

- whenever a name is assigned to a new object, the space held by the prior object is reclaimed if it is not referenced by any other name or object.
- This automatic reclamation of objects' space is known as ***garbage collection***
- Python's garbage collection is based mainly upon ***reference counters***

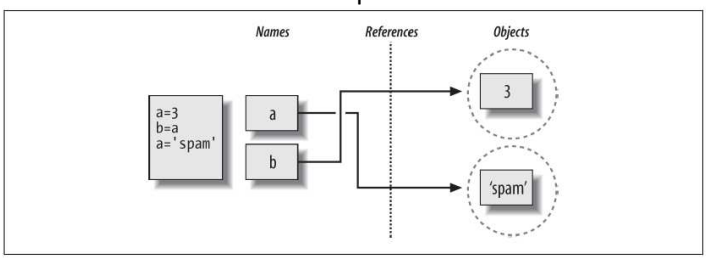
Shared References

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



- This scenario in Python—with multiple names referencing the same object—is usually called a ***shared reference*** (and sometimes just a ***shared object***)

a = 3
b = a
a = 'spam'



a = 3
b = a
a = a + 2

id(): https://docs.python.org/2/library/functions.html#id

Shared References and In-Place Changes

- there are objects and operations that perform ***in-place*** object changes—Python's ***mutable*** **types**, including lists, dictionaries, and sets.
- For objects that support such in-place changes, you need to be more aware of shared references, since a change from one name may impact others.

L1 = [2, 3, 4]
L2 = L1

```
>>> L1 = [2, 3, 4]          # A mutable object
>>> L2 = L1                 # Make a reference to the same object
>>> L1[0] = 24              # An in-place change

>>> L1                      # L1 is different
[24, 3, 4]
>>> L2                      # But so is L2!
[24, 3, 4]
```

Shared References and Equality

- garbage-collection behavior described earlier may be more conceptual than literal for certain types.
- Python caches and reuses small integers and small strings.

x = 42
x = 'shrubbery' # Reclaim 42 now?

- the object 42 here is probably not literally reclaimed; instead, it will likely remain in a **system table** to be reused the next time you generate a 42 in your code.

- there are two different ways to check for equality in a Python program.

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

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

- the **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.
- **'is'** simply compares the pointers that implement references, and it serves as a way to detect shared references in your code if needed.

X = 42
Y = 42
X == Y => True
X is Y => True

- small integers and strings are cached and reused, though, is tells us they reference the same single object.
- the **getrefcount** function in the standard **sys module** returns the **object's reference count**.
- this behavior reflects one of the many ways **Python optimizes** its model for execution speed.

Questions:

1. Consider the following three statements. Do they change the value printed for A?

```
A = "spam"
B = A
B = "shrubbery"
```

2. Consider these three statements. Do they change the printed value of A?

```
A = ["spam"]
B = A
B[0] = "shrubbery"
```

3. How about these—is A changed now?

```
A = ["spam"]
B = A[:]
B[0] = "shrubbery"
```