Dynamic Typing

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

As you know, Python is dynamically typed
- that is, there is no need to really explicit it.

Three separate concepts behind that assignment:
- variable creation, python works out names in spite of the (possible) content
- variable types, no type associated to the variable name, type lives with the object,
- variable use the name is replaced by the object when used in an expression.

What happens inside?
1. create an object to represent the value 42;
   - objects are pieces of allocated memory
2. create the variable a, if it does not exist yet;
   - variables are entries in a system table with spaces for links to objects,
3. link the variable a to the new object 42.
   - references are automatically followed pointers from variables to objects.

Coming from typed languages programming
- this looks as the type of a changes.

Of course, this is not true. In Python
names have no types

We simply changed the variable reference to a different object.

Objects know what type they are.
- Each object has an header field that tells it with its type.
Because objects know their type, variables don’t have to.
**Dynamic Typing**

**Objects Are Garbage-Collected**

What happens during variable reassignment to the value it was referencing?

```python
>>> a = 42
>>> a = 'spam'  # Reclaim 42 now (unless referenced elsewhere)
>>> a = 3.14  # Reclaim 'spam' now
>>> a = [1,2,3]  # Reclaim 3.14 now
```

In Python, the space held by the prior object is reclaimed (garbage collection)
- if it is not referenced by any other name or object

Automatic garbage collection implies less bookkeeping code.

**Dynamic Typing**

**References & Equality**

Two ways to check equality: `==` (equality) and `is` (object identity)

```python
>>> L=[1,2,3]
>>> M=[1,2,3]
>>> N=L
>>> L==M, L
is M
(True, False)
>>> L==N, L
is N
(True, True)
```

But . . .

```python
>>> X=42
>>> Y=42
>>> X==Y, X
is Y
(True, True)
```

Small integers and some other constant objects are cached.

```python
>>> import sys
>>> sys.getrefcount(42)
10
>>> sys.getrefcount([1,2,3])
1
```

**Dynamic Typing**

**Shared References**

What happens when a name changes its reference and the old value is still referred?

```python
>>> a = 42
>>> b = a
>>> a = 'spam'
>>> b
42
```

This is still the same?

```python
>>> X=42
>>> Y=42
>>> X==Y, X
is Y
(True, True)
```

Small integers and some other constant objects are cached.

**Dynamic Typing**

**References & Passing Arguments**

Arguments are passed by value.

```python
X = 42
L = [1,2,3]
def fake_mutable(i,l):
i = i*2
l[1] = '?!?!'
l = {1,3,5,7}
```

```python
from args import fake_mutable, X, L
print("X :- {0} \t L :- {1}".format(X,L))
```

```python
X :- 42 L :- [1, '?!?!', 3]
```

Collections but tuples are passed by reference

```python
>>> L = [1,2,3]
>>> fake_mutable(X,L[::])
```

```python
>>> print("X :- {0} \t L :- {1}".format(X,L))
```

```python
X :- 42 L :- [1, 2, 3]
```

Global values are immutable as well, to change them use `global`

```python
def mutable():
    global X, L
    X = X*2
    L[1] = '?!?!'
    L = (1,3,5,7)
```

```python
if __name__ == '__main__':
mutable()
print("X :- {0} \t L :- {1}".format(X,L))
```
Closures in Action

Currying

\[ f(x, y) = \frac{y}{x} \quad (2.3) \quad g(y) = \frac{y}{2} \quad (2.4) \quad g(3) = \frac{3}{2} \]

```python
def make_currying(f, a):
    def fc(*args):
        return f(a, *args)
    return fc

def f2(x, y):
    return x*y

def f3(x, y, z):
    return x+y+z

if __name__ == '__main__':
a = make_currying(f2, 3)
b = make_currying(f3, 4)
c = make_currying(b, 7)
print('((cf3 4) 7)({0}) :- {1}'.format(5,c(5)))
```

References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
- Mark Lutz.
  Learning Python.
- Mark Pilgrim.
  Dive into Python 3.