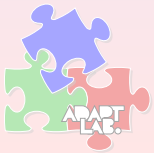
Dynamic Typing

Walter Cazzola

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



Dynamic Typing
Variables, Object and References

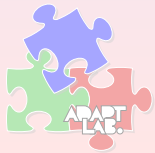
```
[22:55]cazzola@ulik:~/esercizi-pa>python3
>>> a = 42
```

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

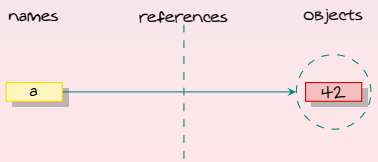


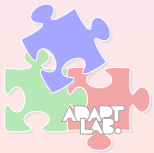
Dynamic Typing
Variables, Object and References

```
[22:55]cazzola@ulik:~/esercizi-pa>python3
>>> a = 42
```

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.





Dynamic Typing
Types Live with Objects, Not Variables

```
[22:57]cazzola@ulik:~/esercizi-pa>python3
>>> a = 42              # it's an integer
>>> a = 'spam'          # now, it's a string
>>> a = 3.14            # now, it's a floating point
```

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 tags it with its type.

Because objects know their type, variables don't have to.

Dynamic Typing
Objects Are Garbage-Collected

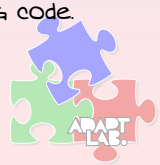
What happens during variable reassignment to the value it was referencing?

```
[22:57]cazzola@ulik:~/esercizi-pa>python3
>>> 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

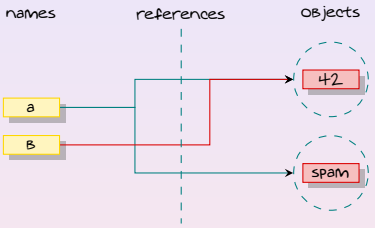
Automatic garbage collection implies less bookkeeping code.



Dynamic Typing
Shared References

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

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



This is still the same?

```
[23:00]cazzola@ulik:~/esercizi-pa>python3
>>> a = [1,2,3]
>>> b=a
>>> b[1]='spam'
>>> b
[1, 'spam', 3]
>>> a
[1, 'spam', 3]
```

Dynamic Typing
References & Equality

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

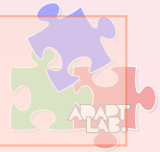
```
[14:59]cazzola@ulik:~/esercizi-pa>python3
>>> L=[1,2,3]
>>> M=[1,2,3]
>>> N=L
>>> L==M, L is M
(True, False)
>>> L==N, L is N
(True, True)
```

But ...

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

Small integers and some other constant objects are cached.

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



Dynamic Typing
References & Passing Arguments

Arguments are passed **by value**.

```
X = 42
L = [1,2,3]

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

```
[18:47]cazzola@ulik:~/esercizi-pa>python3
>>> from args import fake_mutable, X, L
>>> print("X :- {0} \t L :- {1}".format(X,L))
X :- 42          L :- [1, 2, 3]
>>> fake_mutable(X,L)
>>> print("X :- {0} \t L :- {1}".format(X,L))
X :- 42          L :- [1, '?!?!', 3]
```

Collections but tuples are passed **by reference**

```
>>> L = [1,2,3]
>>> fake_mutable(X,L[:])
>>> print("X :- {0} \t L :- {1}".format(X,L))
X :- 42          L :- [1, 2, 3]
```

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

```
def mutable():
   global X, L
   X = X*2
   L[1] = '?!?!'
   L = {1,3,5,7}
if __name__ == "__main__":
   mutable()
   print("X :- {0} \t L :- {1}".format(X,L))
```

```
[19:09]cazzola@ulik:~/esercizi-pa>python3 args.py
X :- 84          L :- {1, 3, 5, 7}
```

Closures in Action
Currying

$$f(x,y) = \frac{y}{x} \xrightarrow{f(2,3)} g(y) = f(2,y) = \frac{y}{2} \xrightarrow{g(3)} g(3) = \frac{3}{2}$$

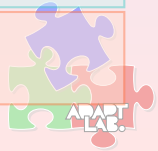
```
def make_currying(f, a):
    def fc(*args):
        return f(a, *args)
    return fc

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("(cf2 3)({0}) :- {1}, (cf3 4)({2},{3}) :- {4}".format(1,a(1),2,3,b(2,3)))
   print("((cf3 4) 7)({0}) :- {1}".format(5,c(5)))
```

```
[19:22]cazzola@ulik:~/esercizi-pa>python3 curry.py
(cf2 3)(1) :- 4, (cf3 4)(2,3) :- 9
((cf3 4) 7)(5) :- 16
```



Look at `partial` in `functools`.

References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
  The Pragmatic Bookshelf, second edition, 2009.
- Mark Lutz.
  Learning Python.
  O'Reilly, third edition, November 2007.
- Mark Pilgrim.
  Dive into Python 3.
  Apress*, 2009.