Extended Introduction to Computer Science CS1001.py

Module A: Python's memory model (take 2), Containers in Python

Instructors: Amir Rubinstein, Michal Kleinbort
Teaching Assistants: Noam Parzanchevsky, Hussen Abu Hamad, Asaf Cassel, Shaked Dovrat

School of Computer Science
Tel-Aviv University
Fall Semester 20-21
http://tau-cs1001-py.wikidot.com/

Reminder: What we did in the previous lecture (Highlights)

- Python's memory model
  - Equality and Identity
  - Mutable vs. immutable types
  - Effects of Assignments

Planned Goals and Topics

1. More on functions and the memory model
   You will understand how information flows through functions.
   You will be motivated to understand Python's memory model.

2. Container types in Python
   - str, list, tuple, dictionary, set

Python's Mechanism for Passing Functions' Parameters

Consider the following function, operating on two arguments:

```
def linear_combination(x,y):
    y = 2*y
    return x+y
```

The formal parameters x and y are local, and their "life time" is just during the execution of the function. They disappear when the function is returned.

Back to Functions: Mechanism for Passing Parameters

```
def linear_combination(x, y):
    y = 2*y
    return x+y
```

Now let us execute it in the following manner

```
>>> a, b = 3, 4     # simultaneous assignment
>>> linear_combination(a,b)
11     # this is the correct value
>>> a
3    # a has NOT changed
>>> b
4    # b has NOT changed
```

The actual parameters, a and b are NOT affected.
The assignment `y=2*y` makes the formal argument y reference another object with a different value inside the body of `linear_combination(x,y)`. This change is kept local, inside the body of the function. The change is not visible by the calling environment.

Memory view for the last example

On the board
or
using PythonTutor (a link to this specific example).

Passing Arguments in Functions' Call

Different programming languages have different mechanisms for passing arguments when a function is called (executed).

In Python, the address of the actual parameters is passed to the corresponding formal parameters in the function.

An assignment to the formal parameter within the function body creates a new object, and causes the formal parameter to address it. This change is not visible to the original caller's environment.

Python Functions: Mutable Objects as Formal Variables

```
def modify_list(lst, i, val):
    '''assign val to lst[i]
    does not return any meaningful value '''
    if i<len(lst):
        lst[i] = val
    return None
```

```
>>> L = [10, 11, 12, 13, 14]
>>> modify_list(L,3,1000)
>>> L
[10, 11, 12, 1000, 14]
```

If the function execution mutates one of its parameters, its address in the function does not change. It remains the same address as in the calling environment. So such mutation does affect the original caller's environment. This phenomena is known as a side effect.

Any changes to the calling environment, which are different than those caused through returned functions' values, are called side effects.

Memory view for the last example

On the board
or
using PythonTutor (a link to this specific example).

Mutable Objects as Formal Parameters: A 2nd Example

Consider the following function, operating on one argument:

```
def increment(lst):
    for i in range(len(lst)):
        lst[i] = lst[i] +1
    # no value returned, same as:  return None
```

Now let us execute it in the following manner

```
>>> list1 = [0,1,2,3]
>>> increment(list1)
>>> list1
[1, 2, 3, 4]      # list1 has changed!
```

In this case too, the formal argument (and local variable) `lst` was mutated inside the body of `increment(lst)`. This mutation is visible back in the calling environment.

Such change occurs only for mutable objects.

Effect of Mutations vs. Assignment inside Function Body

Consider the following function, operating on one argument:

```
def nullify(lst):
    lst = []
    # no value returned, same as:  return None
```

Now let us execute it in the following manner

```
>>> list1 = [0,1,2,3]
>>> nullify(list1)
>>> list1
[0, 1, 2, 3]     # list1 has NOT changed!
```

Any change (like an assignment) to the formal argument, `lst`, that changes the (identity of) the referenced object are not visible in the calling environment, despite the fact that it is a mutable object.

Effect of Mutations vs. Assignment inside Function Body 2

It is possible to detect such changes using id?

```
def nullify(lst):
    print(hex(id(lst)))
    lst = []
    print(hex(id(lst)))
    # no value returned, same as:  return None
```

Now let us execute it in the following manner

```
>>> list1 = [0,1,2,3]
>>> hex(id(list1))
0x1f608f0
>>> nullify(list1)
0x1f608f0
0x11f4918     # id of local var lst has changed
>>> list1
[0, 1, 2, 3]     # (external) list1 has NOT changed!
>>> hex(id(list1))
0x1f608f0
```

Any change (like an assignment) to the formal argument, lst, that changes the (identity of) the referenced object are not visible in the calling environment, despite the fact that it is a mutable object.

Functions: Local vs. Global Variables

Consider the following functions, operating on one argument:

```
def summation_local(n):
    s = sum(range(1,n+1))
    # no value returned
```

Now let us execute it in the following manner

```
>>> s = 0
>>> summation_local(100)
>>> s
0     # s has NOT changed
```

In this example, `s` is local to the function body, and changes are not visible to the original caller of the function.

Functions: Local vs. Global Variables

Consider the following function, operating on one argument:

```
s = 0
def summation_global(n):
    global s
    s = sum(range(1,n+1))
    # no value returned
```

Now let us execute it in the following manner

```
>>> s = 0
>>> summation_global(100)
>>> s
5050     # s has changed!
```

In this example, summation_global declared that it treats s as a global variable. This means that the name s inside the function addresses a variable that is located in the "main" environment. In particular, changes to it do propagate to the original caller of the function.

Functions: Information Flow and Side Effects

To conclude, we saw three ways of passing information from a function back to its original caller:

1. Using return value(s). This typically is the safest and easiest to understand mechanism.
2. Mutating a mutable formal parameter. This often is harder to understand and debug, and more error prone.
3. Via changes to variables that are explicitly declared global. Again, often harder to understand and debug, and more error prone.

2. Container types in Python

- Containers are objects that contain inner elements. We saw 2 such objects so far: str and list.
- There are other useful containers in Python. We can classify them by order and by mutability. Here are the common ones:

	ordered (sequences)	unordered
mutable	list [1,2,3]	set {1,2,3} dict {1:"a", 2:"b", 3:"c"}
immutable	str "abc" tuple (1,2,3)	

Tuples vs. Lists

- Tuples are much like lists, but syntactically they are enclosed in regular brackets, while lists are enclosed in square brackets.

Tuples vs. Lists

- Tuples are much like lists, but syntactically they are enclosed in regular brackets, while lists are enclosed in square brackets.

```
>>> a = (2,3,4)
>>> b = [2,3,4]
>>> type(a)
<class 'tuple'>
>>> type(b)
<class 'list'>
>>> a[1], b[1]
(3, 3)
>>> [a[i]==b[i] for i in range(3)]
[True, True, True]
>>> a==b
False
```

- Tuples are much like lists, only they are immutable.

```
>>> b[0] = 0 # mutating the list
>>> a[0] = 0 # trying to mutate the tuple
Traceback (most recent call last):
    File "<pyshell#30>", line 1, in <module>
    a[0]=0
TypeError:  'tuple' object does not support item assignment
```

Using tuples for function return Values

A function can return more than a single value. For example

```
>>> def mydivmod(a,b):
    ''' integer quotient and remainder of a divided by b '''
    return a//b, a%b
```

When executing this function, we get back two (integer) values, “packed” in a `tuple`.

```
>>> mydivmod(21,5)
(4, 1)
>>> mydivmod(21,5)[0]
4
>>> type(mydivmod(21,5))
<class 'tuple'>
```

Incidentally, the returned values can be assigned simultaneously:

```
>>> d,r = mydivmod(100,7)
>>> d
14
>>> r
2
>>> 7*14+2
100
```

Dictionaries

- Dictionaries contain pairs of elements key:value. They are used as mapping between a set of keys and a set of elements.
- Keys cannot repeat (i.e. they are unique), and must be immutable

```
>>> d = {"France":"Europe", "Genrmay":"Europe", "Japan":"Asia"}
>>> type(d)
<class 'dict'>
>>> d #order of elements not necessarily as defined in initialization*
{'Germany':'Europe', 'France':'Europe', 'Japan':'Asia'}
>>> d["Japan"]
'Asia'
>>> d["Israel"]
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
d["Israel"]
KeyError:  'Israel'
>>> d["Egypt"] = "Africa"
>>> d
{'Germany':'Europe', 'France':'Europe', 'Egypt':'Africa', 'Japan':'Asia'}
```

* changed in version 3.7: Dictionary order is guaranteed to be insertion order. However, it's not a good practice to rely on it because it is version/language dependent.

Sets

- Sets are like dictionaries, but contain elements rather than pairs of key:val.
- In fact they resemble the mathematical notion of a set
- Some examples - in class

```
>>> s = {1,2,3,"a"}
```

Each type in Python supports various operations. For example, str has an operation called title (str.title), list supports count (list.count), etc.
Dictionaries, sets and tuples also support their own operations - we will introduce them when we need them. But don't wait for us!

Before we move on...

Reminder: what to do after each lecture/recitation:

From now until the end of the course, we will learn numerous topics, and present to you the beauty, and challanges, of Computer Science.

We will use Python extensively, and learn new tricks along the way.