Extended Introduction to Computer Science
CS1001.py

Lecture 4: Python's Memory Model; Containers (str, list, tuple, dict, set)

Instructors: Jonathan Berant, Amir Rubinstein

Teaching Assistants: Michal Kleinbort,
Noam Parzanchevsky, Ben Bogin

Founding instructor: Benny Chor

School of Computer Science
Tel-Aviv University
Spring Semester, 2019
http://tau-cs1001-py.wikidot.com

What we did in Lecture 3

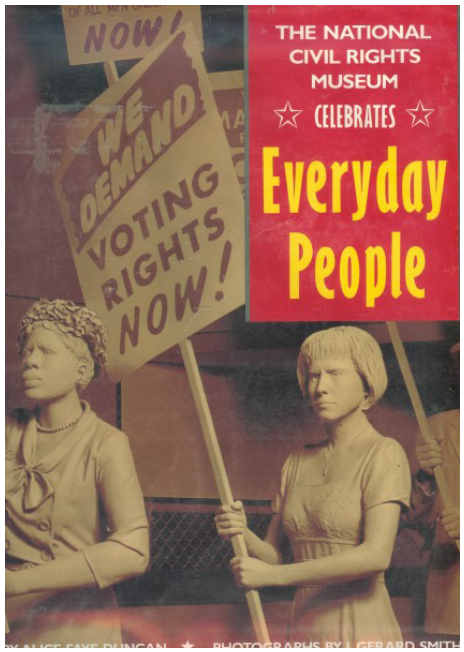
- Lists in Python, list comprehension
- More list and string operators: slicing
- Functions

Lecture 4 - Plan

- Memory model
  - Equality and Identity (the is operator)
  - The id, hex functions
  - Mutable vs. immutable classes
  - Effects of Assignments
  - Understanding Python's memory model when using functions
- Containers in Python

Equality and Identity

Identity, equality, or lack thereof, are of course central issues in society and politics. Struggle over them has shaped and defined nations and societies, as we witness again recently in neighboring countries, here in Israel, and even next to Wall Street:



"The struggle for identity and equality fought by everyday people", pic taken from www.graydogsbooks.com (site does not exist anymore)

These larger issues are, however, out of the scope of our course. We will deal with them only in the context of Python objects.

Equality ≠ Identity in Python

- As we already saw, Python can check equality of an integer and a float.

```
>>> 1 == 1.0
True
```

- Python's interpreter coerces the integer (1) to a float, then checks equality of the two values. In this case, the two values are indeed equal, so it returns True. But are these two objects (numbers) identical? Let us ask Python first:

```
>>> 1 is 1.0
False
>>> 1 is not 1.0
True
```

- These identity operators is and is not examine if the two objects referred to are the same object in memory. As we saw above, identity is a stricter relation than equality (identity implies equality, but not vice versa).

Python's id Function

- Python's interpreter has a built in function, id, which returns the ”identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Using a different terminology, id returns the address of the object in memory.

- Clarification:

  `id(object1) == id(object2)` if and only if `object1 is object2`

- Warning: For optimization reasons, two objects with non-overlapping lifetimes may have the same id value. Furthermore, in two different executions, the same object may be assigned different id. And obviously this is platform dependent.

- When using id, we recommend using the hexadecimal (base 16) representation of the outcome. This obviously is equivalent, yet often more transparent than the decimal representation.
- Do not worry if you are not familiar with the hexadecimal base 16 yet. We will explain it soon.

Python's Memory Model and the id Function

- Variables' names (identifiers) in Python correspond to an object in memory. As a result of a new assignment to an existing variable identifier, a new object (in a different location in memory) is referred to by the same identifier.

- The id allows us to “probe" memory locations directly, and even compare them over time (cannot be done with the is operator).

```
>>> x = 1
>>> id(x)
1494016
>>> hex(id(x))
'0x16cc00'
>>> x = 2
>>> hex(id(x)) # new object, new memory location
'0x16cc20'     # exactly 32 bits away from previous
```

Memory Addresses (cont.)

- The address of an object is typically not uniquely determined by its value:

```
>>> x = 2**200+1
>>> y = 2**200+1
>>> x==y
True
>>> x is y
False
```

- we will now probe the exact memory addresses:

```
>>> hex(id(x))
'0x170d098'
>>> hex(id(y))
'0x170d048'
```

“Small“ Objects

- However, “small“ values, such as small integers up to 256, a few negative integers, and single characters do have a constant memory address along the lifetime of an IDLE session, which is independent of execution history.
- The goal of this is the optimization of memory access.

```
>>> x = 1
>>> hex(id(x))
'0x16cc00'
>>> y = 1
>>> hex(id(y))
'0x16cc00'     # same location as x
>>> x is y
True
```

Lists are mutable

We saw that Python's list is an ordered sequence of elements. Furthermore, list elements have indices, enabling direct (aka “random”) access.

We now ask if lists elements can not just be individually accessed, but also be individually assigned ?

```
>>> list3 = [1,2,3]
>>> list3[2]
3
>>>list3[2] = “Agama stellio”
List3
[1, 2, ‘Agama stellio’]
```

The assignment list3[2] = “Agama stellio” has mutated (changed) the list.

Strings are not Mutable

- Like lists, strings are also indexed (they are "subscriptable"): Individual characters can be directly accessed, using their index. Consider our favorite string, for example:

```
>>> species = "Agama stellio"
>>> species[0]
'A'
>>> species[1]
'g'
>>> species[5]
' '
```

- However, unlike lists, strings are not mutable. Assignment results in an error.

```
>>> species[2] = "t"
Traceback (most recent call last):
     File "<pyshell#4>", line 1, in <module>
          species[2]="t"
TypeError: 'str' object does not support item assignment
```

Numbers are Surely not Mutable

- Mutability means that you can change the content of an object without changing its identity.

- Unlike lists or strings, numbers cannot even be indexed (namely they are not "subscriptable"): we cannot directly access their "inner elements" (bits), nor can we modify them.

- Consequently, numbers (int, float) are not mutable.

The Effect of Assignment

- Let's see another variation

```
>>> x = 257
>>> y = 457-200
>>> z = x
>>> x is y
False
>>> x is z
True
```

- So the effect of the assignment `z = x` is that no new object is created. The only effect is that the variable `z` now refers to the same object as `x`.

Assignments to String Variables

- And now, a few assignments to string variables:

```
>>> course1 = "Intro2CS"
>>> course2 = course1
>>> course1 = "Discrete math"
>>> course2
'Intro2CS'
>>> course1
'Discrete math'
```

- No surprises here either.

Assignments to List Variables

- And a few assignments to list variables:

```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1 = [6,7,8,9]
>>> list2
[1,2,3]
>>> list1
[6,7,8,9]
```

- Still, no surprises (you may start wondering if this discussion is leading anywhere…)

Assignments to List Variables, take 2

- But now look at this - a few assignments to components of list variables:

```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1[0] = 97    # mutating list1
>>> list1
[97,2,3]             # as expected
>>> list2
[97,2,3]             # list2 also mutated!!!
```

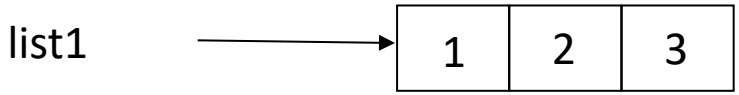
- What the %$*# is happening here?

Assignments vs. Mutation

```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1[0] = 97
```

Assignments vs. Mutation

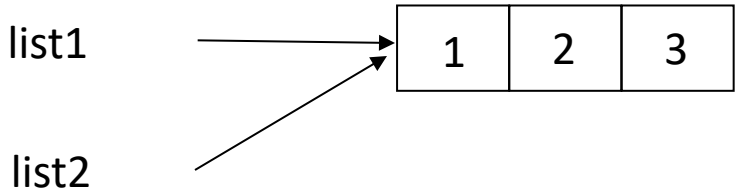
```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1[0] = 97
```



- The assignment list1 = [1,2,3] creates a list object, [1,2,3], and a reference from the variable name, list1, to this object.

Assignments vs. Mutation

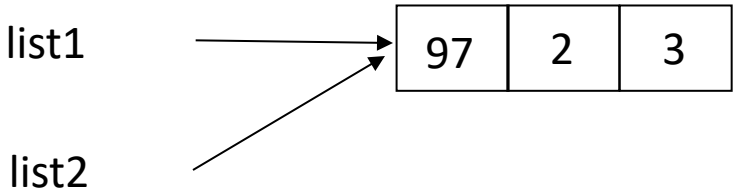
```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1[0] = 97
```



- The assignment list1 = [1,2,3] creates a list object, [1,2,3], and a reference from the variable name, list1, to this object.
- The assignment list2 = list1 does not create a new object. It just creates a new variable name, list2, which now refers to the same object.

Assignments vs. Mutation

```
>>> list1 = [1,2,3]
>>> list2 = list1
>>> list1[0] = 97
```



- The assignment list1 = [1,2,3] creates a list object, [1,2,3], and a reference from the variable name, list1, to this object.
- The assignment list2 = list1 does not create a new object. It just creates a new variable name, list2, which now refers to the same object.
- When we mutate list1[0] = 97, we do not change these references. Thus, displaying list2 produces [97,2,3].

(See this demo on Pythontutor)

Assignments vs. Mutation (cont.)

- For mutable objects, we saw that some “components" of the object can subsequently be changed. This does not change the memory location of the object. For example, mutating a list.

```
>>> list1 = [1,2,3]
>>> hex(id(list1))
'0x290deb8'
>>> list1[0] = 97 # mutating list1, memory location UNCHANGED
>>> list1
[97,2,3]                # mutated indeed
>>> hex(id(list1))
'0x290deb8'             # object memory location UNCHANGED
```

- And now, let's just repeat the first assignment

```
>>> list1 = [1,2,3]
>>> hex(id(list1))
'0x290d968'              # NEW object, new memory location
```

- For mutable objects, like lists, a new assignment to the same identifier with identical value creates a new object with a new address.
- This is in contrast to the “small immutable objects” from before.

One More Look at Mutable Object

- Let us examine lists with identical values yet different addresses.

```
>>> list1 = [1,2,3]
>>> hex(id(list1))
'0x15e9b48'
>>> list2 = list1
>>> hex(id(list2))
'0x15e9b48'             # same same
>>> list3 = [1,2,3]
>>> hex(id(list3))
'0x15e9cb0'             # but different
```

- Now let us see what happens with the components of these lists:

```
>>> list1[0] is list3[0]
True
>>> hex(id(list1[0]))
'0x16cc00'              # looks familiar?
>>> hex(id(list3[0]))
'0x16cc00'              # same as previous
```

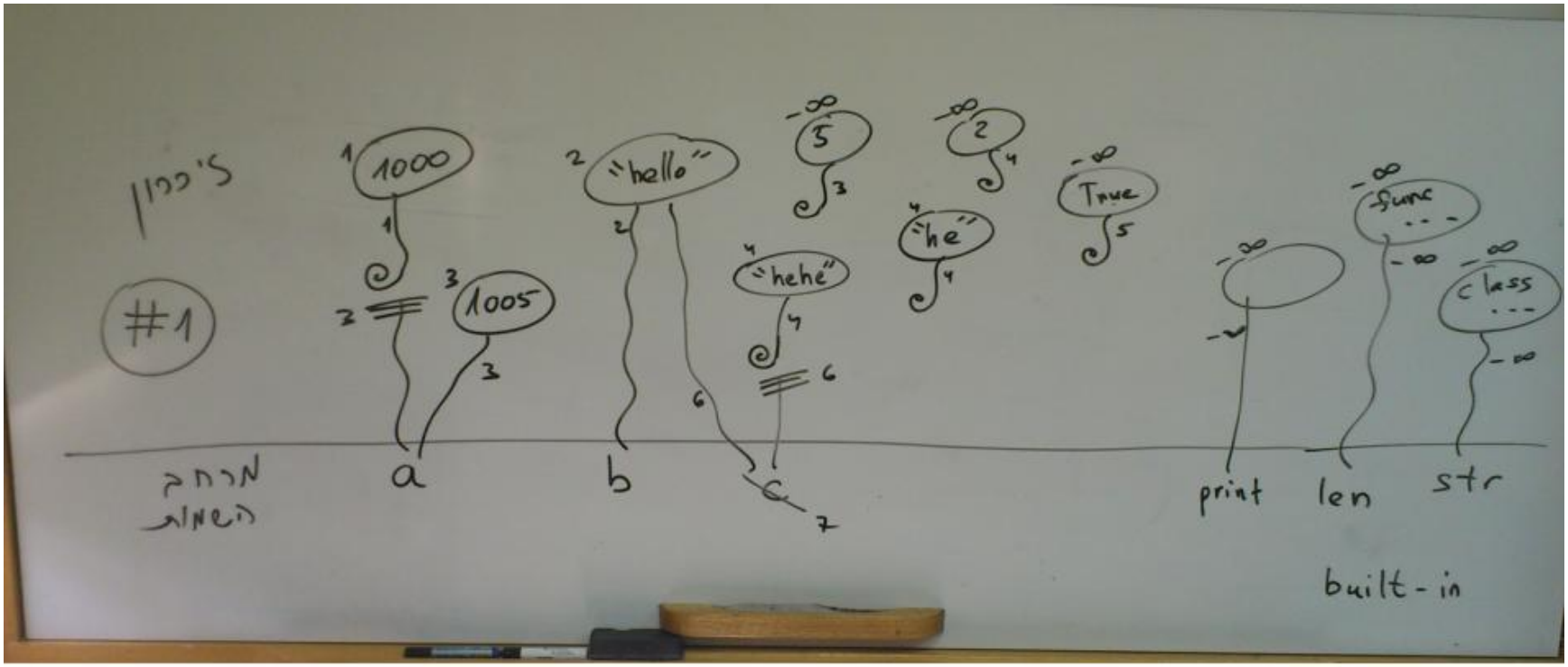
- What graphic images of these lists in memory follow?

Deleting an Object

- So far, we saw that an assignment adds a variable name (if it was not assigned before) and associates an object with it.
- It is also possible to delete a variable. After deletion, the variable no longer exists, and referring to it in an expression yields an error.

```
>>> x = 10
>>> x
10
>>> del x
>>> x
...
NameError: name 'x' is not defined
>>> s = 200
>>> t = s
>>> del s       # s is gone
>>> t           # t is still alive and kicking
200
```

A Graphical View: The Balloons Model



You can use a webtool called Python Tutor to visualize examples (supports python 3.6):
(http://www.pythontutor.com/visualize.html#mode=edit).

Intermediate Summary

- For "large" immutable objects, e.g. large numbers, the address of the object is typically not uniquely determined by the value.
- Positive integers up to 256, a few negative integers, and single characters, do have a single, pre-assigned location in memory.
- Assignment of one variable to another merely creates another reference to the object.
- Mutable objects, such as lists, allow changing their "inner components" without changing the memory location of the "containing" object.
- Python Tutor http://www.pythontutor.com/visualize.html#mode=edit.

More on functions and the memory model

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 four 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.
4. Via `print` commands. This typically is not very problematic. However, it is advisable to separate computations from interface (i.e. communication with the user - print() and input() for example). These two parts are normally independent and are better not interleaved.

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.

```
>>> 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 simultaneously assigned:

```
>>> 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 -
#but in python 3.7 they actually are ordered...
{'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'}
```

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!

3. A comment on operators: a convenient shorthand

Consider the following sequence of instructions:

```
>>> a = 1
>>> a = a+6
>>> a
7
```

Now suppose that, following the advice given by the course staff to give meaningful names to variables, you rewrite the code, using a more meaningful, albeit longer, name:

```
>>> long_cumbersome_name = 1
>>> long_cumbersome_name = long_cumbersome_name + 6
>>> long_cumbersome_name
7
```

Python provides a shorthand for the addition, which may appeal to the young and impatient.

```
>>> long_cumbersome_name = 1
>>> long_cumbersome_name += 6
>>> long_cumbersome_name
7
```

This shorthand is applicable to any assignment where a variable appears on both the right and left hand sides.

A Convenient Shorthand, cont.

This shorthand is applicable to any assignment where a variable appears on both the right and left hand sides.

```
>>> x = 10
>>> x*=4
>>> x
40

>>> x**=2
>>> x
1600

>>> x**=0.5
>>> x
40.0

>>> word = "Dr"
>>> word+=" Strangelove"
>>> word
'Dr Strangelove'
```

Use with some caution: the shorthand is not always equivalent to the original expression (in the "Tirgul").

Before we move on...

Until now we learned mostly Python. Some of you probably feel like this:

Or, desirably, like this:

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.