Extended Introduction to Computer Science CS1001.py

Lecture 4: Functions & Side Effects ; Integer Representations: Unary, Binary, and Other Bases

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University, Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

Reminder: What we did in Lecture 3 (Highlights)

- Additional operations on lists and strings, *e.g. slicing*.
- Functions
- Equality and Identity
- Mutable vs. immutable classes
- Effects of Assignments
- Python's Memory Model
- Un-assignment: Deletion

Lecture 4: Planned Goals and Topics

1. Two additional notes about operators
   - Precedence and associativity.
   - A Convenient Shorthand.

2. 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.
   - Tuples and lists.
   - Multiple values returned by functions.
   - Side effects of function execution.

3. Integer representation
   you will understand how (positive) integers are written in binary and other bases.
   - Natural numbers: Unary vs. binary representation.
   - Natural numbers: Representation in different bases (binary, decimal, octal, hexadecimal, 31, etc.).

Operator Precedence and Associativity

- An expression may include more than one operator
- The order of evaluation depends on operators precedence and associativity.
- Higher precedence operators are evaluated before lower precedence operators.
- The order of evaluation of equal precedence operators depends on their associativity.
- Parentheses override this default ordering.
- No need to know/remember the details: When in doubt, use parentheses!

Operator Precedence and Associativity - Examples

```
>>> 20 - 4 * 3      #  * before - (higher precedence)
8
>>> 20 - (4 * 3)    #   equivalent to the previous one
8
>>> (20 - 4) * 3    #  Parentheses can change the order
48
>>> 3 * 5 // 2      # these equal precedence ops from left to right
7                            # because they are left associative
>>> 3 * (5 // 2)    # Parentheses can change the order
6
>>> 2 ** 3 ** 2     #  ** from right to left (unlike most other ops)
512
>>> (2 ** 3) ** 2   # Parentheses can change the order
64
```

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").

2. More on functions and the memory model

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 inner 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)	

- More details on tuples and dictionaries in the next slides.
- We will encounter sets later on in the course.

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, 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'}
```

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
>>> b
4    # b has NOT changed
```

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

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 len(lst)>i:
        lst[i]= val
    return None
```

```
>>> lst = [0,1,2,3,4]
>>> modify_list(lst,3,1000)
>>> lst
[0, 1, 2, 1000, 4]
```

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

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
```

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
```

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
```

Now let us execute it in the following manner

```
>>> list1=[0,1,2,3]
>>> hex(id(lst))
0x1f608f0
>>> nullify(list1)
0x1f608f0
0x11f4918     # id of local var lst has changed
>>> list1
[0, 1, 2, 3]     # (external) list1 has NOT changed!
>>> hex(id(lst))
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):
    partial_sum = sum(range(1,n+1))
# no value returned
```

Now let us execute it in the following manner

```
>>> partial_sum = 0
>>> summation_local(5)
>>> partial_sum
0     # partial_sum has NOT changed
```

In this example, `partial_sum` 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:

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

In this example, the declaration `global partial_sum` makes this variable global. In particular, changes to it do propagate to the original caller of the function.

Functions: Local vs. Global Variables

Lets run both versions ("local" and "global"):

```
>>> partial_sum
Traceback (most recent call last):
File "<pyshell#11>", line 1, in <module>
partial_sum
NameError:  name 'partial_sum' is not defined
>>> summation_local(5)
>>> partial_sum
Traceback (most recent call last):
File "<pyshell#11>", line 1, in <module>
partial_sum
NameError:  name 'partial_sum' is not defined
>>> summation_global(5)
>>> partial_sum
15
```

The declaration `global partial_sum` makes this variable global. In particular, changes to it do propagate to the original caller of the function (including defining it).

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.

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.

2. Integer representation

Natural Numbers

"God created the natural numbers; all the rest is the work of man"
(Leopold Kronecker, 1823 –1891).

In this course, we will look at some issues related to natural numbers. We will be careful to pay attention to the computational overhead of carrying the various operations.

Today:

- Integers: Unary vs. binary representation.
- Integers in Python.
- Representation in different bases (binary, decimal, octal, hexadecimal, 31, etc.).

In the (near) future:

- Addition, multiplication and division.
- Exponentiation and modular exponentiation.
- Euclid's greatest common divisor (gcd) algorithm.
- Probabilistic primality testing.
- Diffie-Helman secret sharing protocol.
- The RSA public key cryptosystem.

Python Floating Point Numbers (reminder)

But before we start, we wish to remind that Python supports three types of numbers. The first type, `float`, is used to represent real numbers (in fact these are rational, or finite, approximations to real numbers). For example, $1.0, 1.322, \pi$ are all floating point numbers, and their Python type is `float`. Representation of floating point numbers is discussed at a later class (and in more detail in the computer structure course (second year).

```
>>> type(1.0)
<class 'float'>
>>> import math
>>> math.pi
3.141592653589793
>>> type(math.pi)
<class 'float'>
```

Remark: Python has an extensive standard library. Some functions are very useful and are always available (*e.g.*, `print, len, max`); Others need "an invitation". To use the functions (constants, and classes...) grouped together in the module `xyz`, type `import xyz`.

Python Integers: Reminder

The second type of numbers are the integers, including both positive, negative, and zero. Their type in Python is `int`

```
>>> type(1),type(0),type(-1)
    # packing on one line to save precious real estate space
(<class ‘int’>,<class ‘int’>,<class ‘int’>)
```

Python Complex Numbers

The third type of numbers are the complex numbers

```
>>> complex(1,1)
(1+1j)
>>> type(complex(1,1))
<class 'complex'>     # complex numbers class
>>> 1j**2      # ** is exponentiation; $(\sqrt{-1})^2$ here
(-1+0j)          # $(\sqrt{-1})^2 = -1$
>>> import math
>>> math.e**(math.pi*(0+1j))
-1+1.2246467991473532e-16j
>>> type(math.e**(math.pi*(0+1j)))
<class 'complex'>
```

Deep Inside the Computer Circuitry

Most of the logic and memory hardware inside the computer are electronic devices containing a huge number of transistors (the transistor was invented in 1947, Bell Labs, NJ). At any given point in time, the voltage in each of these tiny devices (1 nanometer $= 10^{-9}$ meter) is either $+5$v or 0v. Transistors operate as switches, and are combined to complex and functional circuitry.

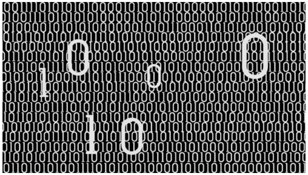
An extremely useful abstraction is to ignore the underlying electric details, and view the voltage levels as bits (binary digits): 0v is viewed as 0, $+5$v is viewed as 1.

From Bits to Numbers and Beyond - Hierarchy of Abstraction Levels

An extremely useful abstraction is to ignore the underlying electric details, and view the voltage levels as bits (binary digits): 0v is viewed as 0, +5v is viewed as 1.

The next conceptual step is arranging these bits so they can represent natural numbers.

Then, we will strive to arrange natural numbers so they can represent other types of numbers - negative integers, real numbers, complex numbers, and furthermore characters, text, pictures, audio, video, etc. etc.



(figure taken from http://www.learner.org)

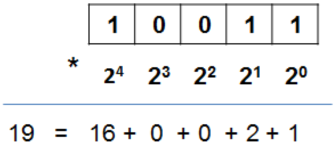
We will begin at the beginning: From bits to numbers.

Unary vs. Binary Representation of Numbers

Consider the natural number nineteen.

In unary, it is represented as 1111111111111111111.
In binary, it is represented as 10011.

The two representations refer to the same entity (the natural number nineteen, written in decimal as 19).



	1	0	0	1	1
*	2^4	2^3	2^2	2^1	2^0

19 = 16 + 0 + 0 + 2 + 1

However, the lengths of the two representations are substantially different. The unary representation is exponentially longer than the binary representation.

To see this, consider the natural number 2^n. In unary it is represented by 2^n ones. In binary, it is represented by a single one, followed by n zeroes.

Representation of Natural numbers in an arbitrary base

A natural number N can be represented in a base b ($b > 1$, an integer), as a polynomial, whose coefficients are natural numbers smaller than b.

The coefficients of the polynomial are the digits of N in its b-ary representation.

$$N = a_k * b^k + a_{k-1} * b^{k-1} + ... + a_1 * b + a_0$$

where for each i , $0 \leq a_i \lneqq b$.

$$N_{in\ base\ b} = (a_k a_{k-1} ..... a_1 a_0)$$

Claim: The natural number N represented as a polynomial of degree k (has $k+1$ digits) in base b satisfies $b^k \leq N \lneqq b^{k+1}$.
Can you prove this?

Unary, Binary and Other Representations

Beside the commonly used decimal (base 10) and binary (base 2) representations, other representations are also in use. In particular the ternary (base 3), octal (base 8) and hexadecimal (hex, `0x`, base 16) are well known.

The lengths of representations in different bases differ. However, the lengths in bases $b \geq 2$ and $c \geq 2$ are related linearly. For example, a number represented with d digits (in decimal) will take at most $\lceil d \log_2 10 \rceil$ bits (in binary).

Can you prove this?

As alluded to earlier, natural numbers in the computer are represented in binary, for a variety of reasons having to do with the physical world, electric circuits, capacities, currents, etc..

Different Bases Representations in Python

Python has built in functions for converting a number from decimal (base 10) to binary, octal (base 8), and hexadecimal (base 16).

The returned values are strings, whose prefixes `0b,0o,0x` indicate the bases $2, 8, 16$, respectively.

```
>>> bin(1000)
‘0b1111101000’
>>> oct(1000)
‘0o1750’
>>> hex(1000)
‘0x3e8’ #hexadedimal digits are 0,1,2,...,9,a,b,c,d,e,f

>>> type(bin(1000))
<class 'str'>
```

Hexadecimal Representations in Python

In hex, the letters `a,b,...,f` indicate the “digits” $10, 11, \ldots, 15$, respectively.

```
>>> hex(10)
'0xa'
>>> hex(15)
'0xf'
>>> hex(62)
'0x3e'          # 62 = 3*16 + 14
```

Recitation ("tirgul"): Conversion to “target bases” $\neq 2, 8, 16$.

Converting to Decimal in Python

Python has a built-in function, int, for converting a number from base b representation to decimal representation. The input is a string, which is a representation of a number in base b (for the power of two bases, `0b, 0o, 0x` prefixes are optional), and the base, b itself .

```
>>> int("0110",2)
6
>>> int("0b0110",2)
6
>>> int("f",16)
15
>>> int("fff",16)
4095
>>> int("fff",17)
4605
>>> int("ben",24)
6695
>>> int("ben",23)  # "a" is 10,...,"m" is 22, so no "n" in base 23
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    int("ben",23)
ValueError:  invalid literal for int() with base 23:'ben'
```

Natural Numbers and Computer Words

Modern computers are arranged by words, groups of fixed number of bits that correspond to size of registers, units of memory, etc. Word size is uniform across a computer, and depends both on the hardware (processors, memory, etc.) and on the operating system.

Typical word sizes are 8, 16 (Intel original 8086), 32, or 64 bits (most probably used by your PC or iMAC).

In many programming languages, integers are represented by either a single computer word, or by two computer words. In Java, for example, these are termed `int` and `long`, correspondingly. Things are quite different in Python, as we will soon see.

Natural Numbers and Integers in Python

A 32 bits word can represent any integer, k, in the range $-2^{31} \leq k \leq 2^{31} - 1$.
To handle larger numbers, these 32 bits words should be manipulated correspondingly (represent signed sums of successive powers of 2^{32}).
This is either done explicitly by the user/programmer, or provided directly by the programming language.

Python takes care of large integers internally, even if they are way over the word size.

```
>>> 3**97-2**121+17

19088056320749371083854654541807988572109959828
```

Natural Numbers and Integers in Python, cont.

Still, when manipulating large integers, one should think of the computational resources involved:

- Space: There is no difficulty to represent $2^{2^{10}} - 17$, or, in Python `2**(2**10)-17` (as the result takes four lines on my Python Shell window, I skip it). But don't try $2^{2^{100}} - 17$ (why?)!
- Time: How many basic operations (or, clock cycles) are needed?
- Time will grow as n, the number of bits in the numbers we operate on, grow. How time grows as a function of n is important since this will make the difference between fast and slow tasks, and even between feasible and infeasible tasks.
- We will define these notions precisely in the future, but you should start paying attention today.

Positive and Negative Integers in the Computer (for reference only)

On any modern computer hardware, running any operating system, natural numbers are represented in binary.

An interesting issue is how integers (esp. negative integers) are represented so that elementary operations like increment/decrement, addition, subtraction, and negation are efficient. The two common ways are

1. One's complement, where, *e.g.* -1 is represented by 11111110, and there are two representations for zero: $+0$ is represented by 00000000, and -0 is represented by 11111111.
2. Two's complement, where, *e.g.* -1 is represented by 11111111, -2 by 11111110, and there is only one 0, represented by 00000000.

Two's complement is implemented more often.

Two's Complement (for reference only)

For the sake of completeness, we'll explain how negative integers are represented in the two's complement representation. To begin with, non negative integers have 0 as their leading (leftmost) bit, while negative integers have 1 as their leading (leftmost) bit.

Suppose we have a k bits, non negative integer, M.
To represent $-M$, we compute $2^k - M$, and drop the leading (leftmost) bit.
For the sake of simplicity, suppose we deal with an 8 bits number:

```
100000000                  100000000
 00000001     M = 1         00001110     M = 14
---------                  ---------
 11111111     -1            11110010     -14
```

A desirable property of two's complement is that we add two numbers "as usual", regardless of whether they are positive or negative.