

Extended Introduction to Computer Science

CS1001.py

Module A Python Basics (cont.):

Lecture 3 Lists, Functions, Memory Model

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
<http://tau-cs1001-py.wikidot.com>

* Slides based on a course designed by Prof. Benny Chor

עדכונים קצרים

- מודל חסם את האפשרות להגיש קבצי `py` מסיבות אבטחה. נעדכן בקרוב כיצד להגיש.

Last Time

- **Conditional** statements including nested ones
- Type `bool` (**Boolean**)
 - The values `True` and `False`
 - **logical** operators (`and`, `or`, `not`)
 - **comparison** operators (`==`, `!=`, `<`, ...)
- Loops (`while`, `for`)
- Collections
`str`, `range`, `list`

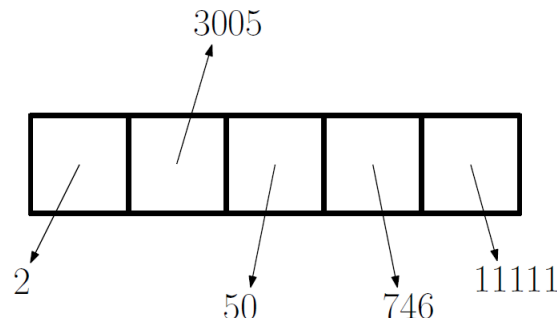
This Lecture

- More on lists: list comprehension
- Functions
- Python's memory model

Type list (reminder)

- `str` in Python is a **sequence** (ordered collection) of **characters**
- `range` in Python is a **sequence** (ordered collection) of **integers**
- `list` in Python is a **sequence** (ordered collection) of **elements** (of any type)
- The simplest way to create a list in Python is to enclose its elements in **square brackets**:

```
>>> my_list = [2, 3005, 50, 746, 11111]
>>> my_list
[2, 3005, 50, 746, 11111]
```



Lists (and strings) are Indexable (reminder)

- Elements of lists and strings can be accessed via their **position**, or **index**. This is called **direct access** (aka “**random access**”)
- In this respect, lists are similar to **arrays** in other programming languages (yet they differ in other aspects)
- Indices in Python **start with 0**, not with 1

```
>>> my_list = [2, 3005, 50, 746, 11111]
>>> my_list[0]
2
>>> my_list[4]
11111
>>> my_list[5]
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    my_list[5]
IndexError: list index out of range
```

Iterating over Lists: Example

```
L = [1, 2, 3, 4]
```

```
product = 1
```

```
for k in L:
```

```
    product = product * k
```

```
print(product)
```

```
???
```

What would happen if we initialized product to 0 instead of 1?

“same as”:

```
for i in range(len(L)):
```

```
    product = product * L[i]
```

Ways to Generate Lists

1) Explicit: `[1, 11, 21, 31]`

2) Via loop:

```
L = []  
for i in range(40):  
    if i%10 == 1:  
        L = L + [i]
```

3) Slicing an existing list

```
L = [1, 11, 21, 31, 41, 51, 61]  
L = L[0:4]
```

4) Direct casting of other sequences: `L = list(range(1, 40, 10))`

5) List comprehension: `L = [i for i in range(40) if i%10==1]`

List Comprehension

- In **mathematics**, concise notations are often used to express various constructs, such as sequences or sets.
- For example, $\{n^2 \mid 1 \leq n < 10 \wedge n \text{ is odd}\}$ is the set of squares of odd numbers that are smaller than 10 (the numbers, not the squares). This set is $\{1, 9, 25, 49, 81\}$
- Python supports a mechanism called **list comprehension**, which enables syntactically concise ways to create lists.
For example:

```
>>> [n**2 for n in range(1,10) if n%2 == 1]
[1, 9, 25, 49, 81]
```

- **Syntax:** `[expression for variable in collection if condition]`

List Comprehension (cont.)

- List comprehension is a powerful tool, allowing the succinct, powerful creation of **new** lists from other collections.

```
>>> staff = ["amir", "michal", "matan", "sapir", "dror", "shahar"]
```

```
>>> L = [st for st in staff if st[0]=="m"]
```

```
>>> L
```

```
["michal", "matan"]
```

```
>>> [st.replace("m", "M") for st in staff if st[0]=="m"]  
["Michal", "Matan"]
```

```
>>> print(staff)
```

```
["amir", "michal", "matan", "sapir", "dror", 'shahar']
```

(the original list is unchanged)

Comic Relief*

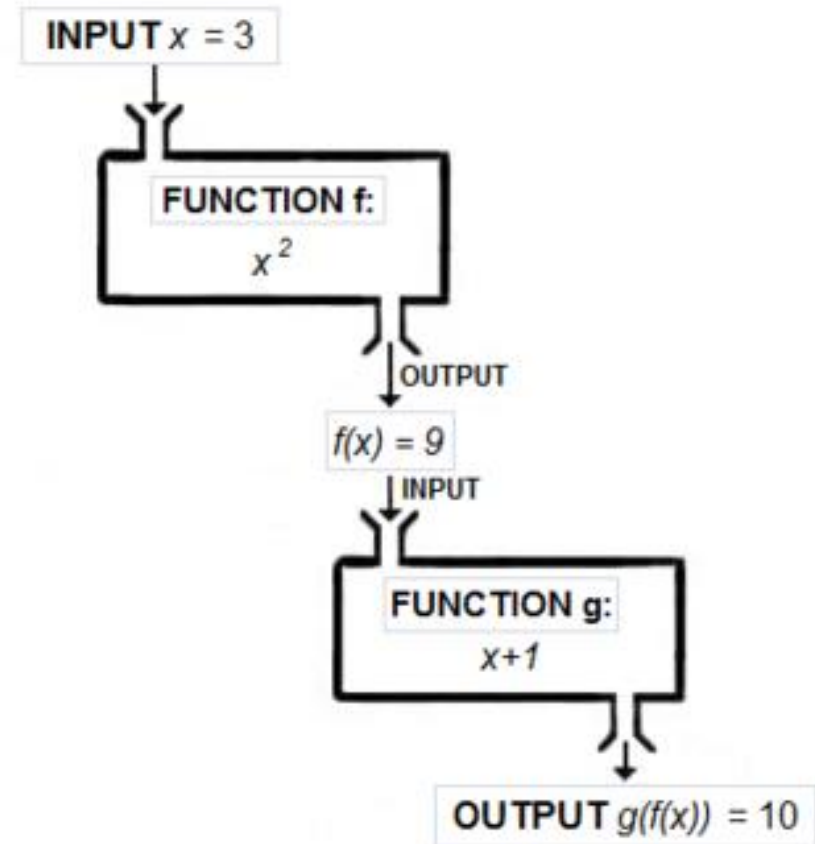
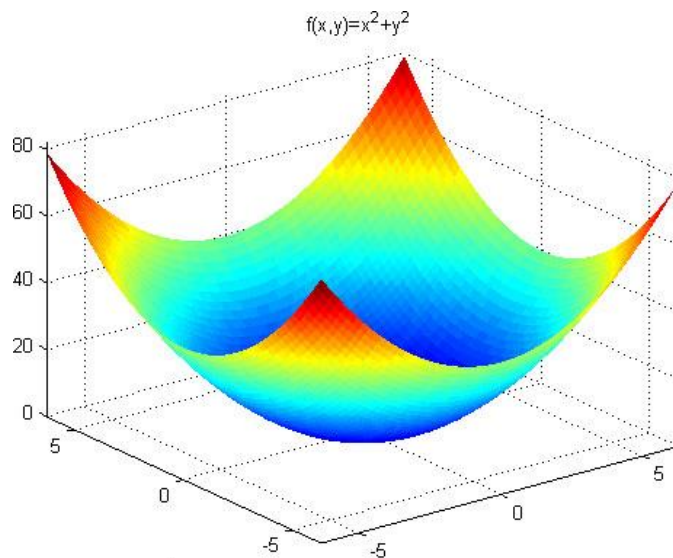
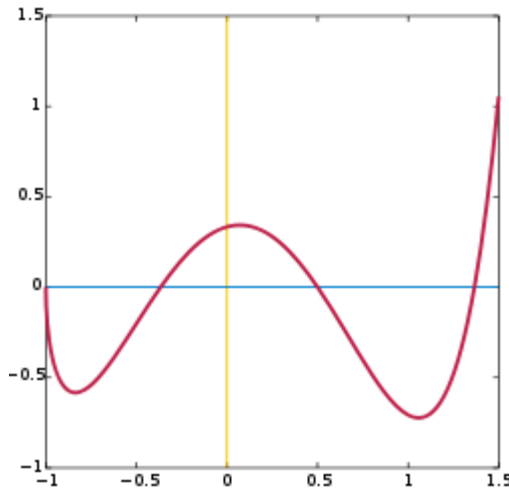
“The best way to learn a language is to speak to natives.”

The guy learning python:



* אני מזמין אתכם לשלוח לי הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Functions



(Figures taken from Wikipedia and (the colorful one) from www.mathworks.com site.)

Function Terminology

- A **function**, in a mathematical sense, expresses the idea that one quantity (the **argument** of the function, also known as the **input**) completely determines another quantity (the **value**, or the **output**).
- A function assigns **exactly one value** to each input.
- For each argument value x , the corresponding unique y is called the **function value** at x , the **output** of f for an argument x , or the **image** of x under f . The image of x may be written as $f(x)$.

Built-in Functions in Python

```
>>> res = sum([1,2,3])
```

```
>>> res
```

```
6
```

```
>>> res = sorted([3,1,2])
```

```
>>> res
```

```
[1,2,3]
```

```
>>> res = str.find("Same but different", "e")
```

```
>>> res
```

```
3
```

- All these and many other are already **implemented** for us and ready to use. But what about designing new, **user-defined** functions?

User-defined Functions

- We saw before a variant of the following piece of code for computing **XOR** (exclusive OR):

```
>>> (a and (not b)) or ((not a) and b)
...
>>> (c and (not d)) or ((not c) and d)
...
```

- This is annoying and time consuming to write and rewrite the same expression, only with different values each time.
- A **function** will come in handy: it will serve as a **template**

A Function for XOR

- Function definition:

```
def xor(x, y):  
    return (x and (not y)) or ((not x) and y)
```

- Function call:

```
>>> xor(True, True)  
False  
>>> xor(True, False)  
True  
  
>>> a = True  
>>> b = False  
>>> xor(a,b)  
True  
>>> xor(True, 3<4)  
False
```


Functions **Definition** Syntax in Python

Formal parameters



```
def xor(x, y):  
    return (x and (not y)) or ((not x) and y)
```

- The keyword `def` indicates the beginning of a function **definition**.
- The function name, in our case `xor`, follows.
- Then, in parenthesis, are the **formal parameters** (`x`, `y` in our case).


There could be **zero** or more formal parameters.

- This ends in a colon (`:`), indicating the beginning of the **function body**.
- Following the colon, the body is, as usual, **indented** by a tab.
- The value following the `return` keyword is the **returned value** of the function.

Functions **Call** Syntax in Python

```
>>> a = True
>>> b = True
>>> xor(a, b)
False
>>> xor(a, False)
True
```

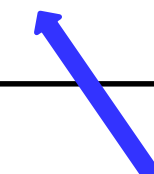
Actual parameters



- A function is **called** by specifying its name, followed by **actual parameters**, or **arguments**.
- The number of **actual** parameters is identical to the number of **formal** parameters (there are some exceptions to this rule, which will be pointed out at a later stage).
- The actual parameters are evaluated and passed to the body of the function to be executed.

Documenting Python's Functions

```
def xor(x, y):  
    ''' computes xor of two Boolean variables, x and y '''  
    return (x and (not y)) or ((not x) and y)
```



```
>>> help(xor)
```

```
Help on function xor in module __main__ :
```

```
xor(x, y)
```

```
    computes xor of two Boolean variables, x and y
```

docstring

- Python provides a mechanism to **document functions**. This is text (possibly multi line, unlike a comment) between **triple quotes**, called a **docstring**. (Three single quotes or three double quotes.)
- Everything between the start and end of the triple quotes is part of a single string, including carriage returns and other quote characters. You can use triple quotes anywhere, but they are most often used when defining **docstrings**.
- When typing help with the function name (in parenthesis), the function's docstring is printed.

(Some of these explanations are taken from "dive into Python", section 2.3)

Functions with “no returned value”

```
def double(x):  
    """ prints twice the value of x """  
    print(2*x)
```

- Note that this function **contains no return statement!**
- Consequently, it returns the special value `None`.
- The same would happen if we wrote just `return` followed by nothing, or even `return None` (both considered safer than omitting the return statement)

No Returned Value (cont.)

- See what happens when assigning `res = twice(...)`, then asking the interpreter for the value of `res`.

```
>>> twice(15)
30
>>> res = twice(15)
30
>>> print(res)
None
>>> type(res)
<class 'NoneType'>
```

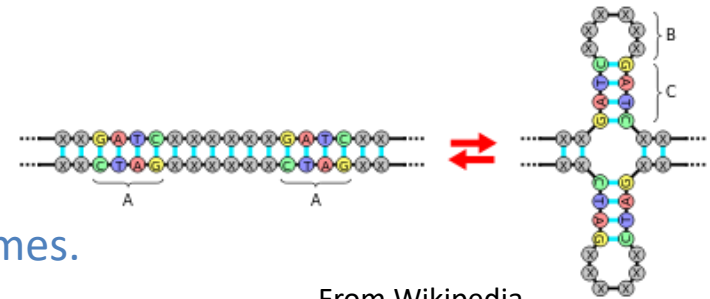
This is a result of the
print inside twice,
NOT a return value

- Python's `print` is another such example:

```
>>> res = print("hi")
hi
>>> res == None
True
```

Another Example: Palindromes

- A **palindrome** is a string w satisfying $w = w^R$ (the string equals its **reverse**).
 - "Madam I'm Adam"
 - "Dennis and Edna sinned"
 - "Red rum, sir, is murder"
 - "Able was I ere I saw Elba"
 - "In girum imus nocte et consumimur igni" (Latin: "we go into the circle by night, we are consumed by fire".)
- And yes, we have **cheated** a bit by ignoring spaces as well as lower/upper case.
- By the way, palindromes also appear in **nature**.
For example as **DNA restriction sites** -
short genomic strings over $\{A,C,T,G\}$
being cut by (naturally occurring) **restriction enzymes**.



From Wikipedia

Another Example: Palindromes

- מתוך ויקיפדיה :
 - ילד כותב בתוך דלי
 - נתנו תואר לאברהם, מהר בא לראותו נתן
 - כולם לא בשיאם רק רמאי שבא למלך
 - דעו מאביכם כי לא בוש אבוש, שוב אשוב אליכם כי בא מועד
- According to [Guinness World Records](#), the Finnish 19-letter word *saippuaktivikauppias* (a soapstone vendor), is the world's longest palindromic word in everyday use

Identifying Palindromes

- What we want now is to write down a program that on input **word**, a string, checks if **word** is a **palindrome**. If it is, the program should return True. If not, it should return False.
- How should we go about writing such program?
- Plan:
 - go over string **characters from both ends** “in parallel” and compare them.
 - As soon as an **inequality** is found, we conclude this is **not a palindrome** and return **False** (skipping the function's remaining execution).
 - If **all** checks found **equality**, we conclude this **is a palindrome**.

Planning the Algorithm

Algorithm: Check if a given string is a palindrome

Input: **word** (type string) of size **n**

Output: **True** is palindrome, **False** otherwise

1. left = 0
2. right = n-1
3. while left < right:
 - 3.1 if word[left] ≠ word[right]
 - 3.1.1 terminate with **False**
 - 3.2 left = left+1
 - 3.3 right = right-1
4. terminate with **True**



Pseudo-code:

not executable, but
clear and does not
assume knowledge of
a specific programming
language


Implementing the Algorithm

```
def is_palindrome(word):  
    ''' checks if word is a palindrome '''  
    n = len(word)  
    left = 0  
    right = n-1  
  
    while left < right:  
        if word[left] != word[right]: mismatch, not palindrome  
            return False  
        left = left+1  
        right = right-1  
  
    return True          # matches all the way, a palindrome
```

Adding “Diagnostic” Printouts

- If you have difficulties following what is going on inside the loop, it may be helpful to print **intermediate results**.

```
def is_palindrome(word):  
    ''' checks if word is a palindrome '''  
    n = len(word)  
    left = 0  
    right = n-1  
  
    while left < right:  
        if word[left] != word[right]: mismatch, not palindrome  
            print("mismatch at indices", left, right)  
            return False  
        left = left+1  
        right = right-1  
    return True          # matches all the way, a palindrome
```



The Three C's*

- Correctness:
 - is it correct?
 - what are the special cases?
- Complexity:
 - Is it efficient enough?
 - can we improve?
- Clarity
 - Can we write it simpler or “nicer” at no significant cost?
 - Is the code easy to modify / extend?

Tests for Checking Correctness

- Let us run the palindrome checking function on several inputs:

```
>>> is_palindrome("ab")
False
>>> is_palindrome("aa")
True
>>> is_palindrome("99899")
True
>>> is_palindrome("998 99")
False
>>> is_palindrome("x")
True
>>> is_palindrome("")
True
>>> is_palindrome(99899)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    is_palindrome(98899)
  File "C:\Users\amiramy\Documents\amiramy\IntroCS\introCSfall2013\Lec
    3\palindrome.py", line 3, in is_palindrome
    n = len(word)
TypeError: object of type 'int' has no len()
```

- When testing your program think of as many special cases as possible
- The above tests are far from providing a **proof** for the function's **correctness**.
- In this case we can **prove** the function correctness using **induction** on the **input size**
- The field of **software verification** provides additional formal approaches for such proofs.

Complexity: Run-Time Analysis

- Suppose our input, `word`, is a string of length n . This means it is composed of n characters.
- How many iterations will the function take to execute on `word`?
- The answer to this depends not only on the length of word, but also to a large extent on word itself:
 - In the best case (from runtime point of view), `word[0]` and `word[n-1]` are not equal. In this case, the execution will terminate after one comparison.
 - In the worst case (again, from runtime point of view), `word` is a palindrome. In this case, the execution will terminate after exactly $n//2$ comparisons (no matter if n is odd or even).
 - You may wonder what will the average case looks like. To answer it, we should know something about the distribution of inputs. We will not tackle this question right now.

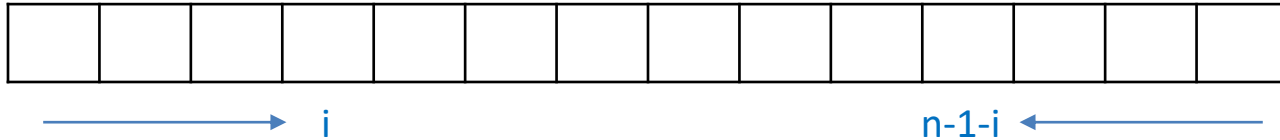
Clarity:

Another Version for Identifying Palindromes

```
def is_palindrome2(word):  
    ''' checks if word is a palindrome '''  
    return word == word[::-1] # does word = its reverse
```

- This code is definitely slimmer and clearer than the previous one. Also, it is almost trivially correct
- What about complexity? Is it also better performance wise?

Yet another Version for Identifying Palindromes



- Iterate over every index `i` checking if `word[i] ≠ word[(n-1)- i]`
- Now it's easier with a `for` loop

```
def is_palindrome3(word):  
    ''' checks if word is a palindrome '''  
    n = len(word)  
  
    for i in range(n):  
        if word[i] != word[n-1-i]: # mismatch, no palindrome  
            return False  
  
    return True                    # matches all the way, a palindrome
```


Improving Worst-Case Efficiency

- As you may have figured out, the last version compares matching positions twice! we could (and should) **reduce** the number of **iterations** in the worst case by **~50%**.
- Which of these is a correct solution, and the most efficient one?

1) `for i in range (n//2 - 1) :`

2) `for i in range (n//2) :`

3) `for i in range (n//2 + 1) :`

- Think first, then check...

Comic Relief*

Aibohphobia is the fear of palindromes. A palindrome is a word that is spelt the same backwards as it is forward.

* אני מזמין אתכם לשלוח לי הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Python's Memory Model

- Will be done mostly **interactively** in class. Slides that cover this are also available in the course site.
- The next slides summarize what we will see

Python's Memory Model

- Objects are stored at specific **memory locations**
`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**.
- **Variables** are temporary names for memory addresses
- Memory address does **not** imply value, Value does **not** imply memory address (except for “**small**” integers, and some strings, for optimization)
- 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>

Python's Mechanism for Passing Functions' Parameters

- Different programming languages have different mechanisms for passing arguments when a function is called (executed).
- In Python, the **address** of the **actual parameter** 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**.
- However, when the function execution **mutates** one of its parameters, its **address** in the function does **not change**, and it remains in 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**.

Information Flow and Side Effects of functions

- 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.

Lecture 3: Highlights

- List comprehension
- Functions
 - Definition, formal parameters
 - Call, actual parameters
 - return value
- “The three C’s”
 - Correctness
 - Complexity
 - Clarity
- Python’s Memory Model
 - Equality vs. identity
 - Mutable vs. immutable types, assignment vs. mutation
 - Function call mechanism (by address)