

Extended Introduction to Computer Science

CS1001.py

Chapter A Acquaintance with Python

Lecture 2 (cont.)

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Fall Semester 2023-4
<http://tau-cs1001-py.wikidot.com>

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

עדכונים קצרים

- ת"ב 1 פורסם
- שעות הקבלה והחונכות העדכניות - בקרוב באתר
- שעת הקבלה שלי (אמיר) עד הודעה חדשה: ראשון 16-17
 - אם יהיה ביקוש אקיים שעת קבלה בזום בהמשך השבוע או ביום שישי
- לאור פניה ממשרתי מילואים: נעשה מאמץ להעלות הקלטות השיעורים עד סוף היום בו התקיים השיעור (לצערנו לעיתים ייתכנו עיכובים בשל עניינים טכניים).

Last Time

- Programing (high level) language → machine code
- IDLE
- Python basics:
 - Types (int, float, str)
 - Variables
 - operators
 - ~~Conditionals~~ (but you saw it in the recitations)

This Lecture

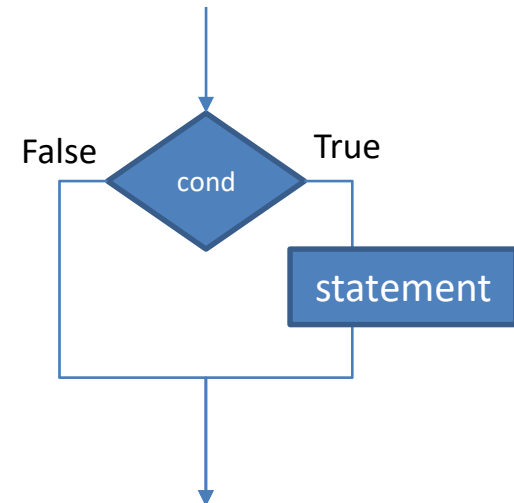
- Conditional statements
- More on variables, types and operators
 - Type `bool` (Boolean)
 - Logical operators (`and`, `or`, `not`)
 - Comparison operators (`<`, `<=`, `>`, `>=`, `==`, `!=`)
- Loops (`while`, `for`)
- Collections
 - (Type `str`)
 - Type `range`
 - Type `list`

Conditional Statements

- The **flow** of very simple programs is **linear**: we execute one statement after the next, in the order they were typed.
- However, the flow of most programs depends on values of variables and relations among them. Conditional statements are used to **split** this flow.

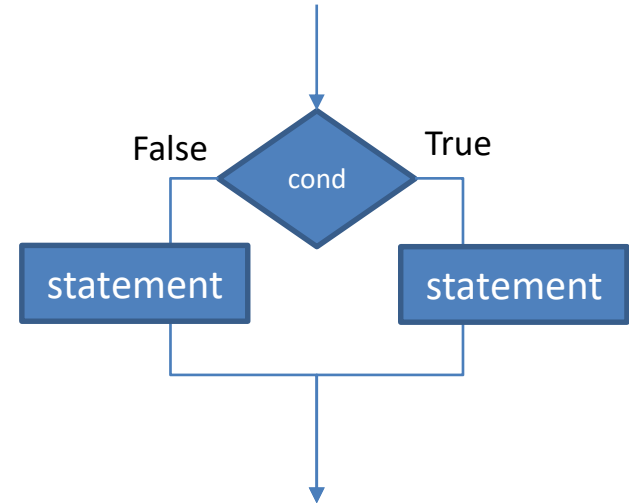
```
melting = 0
boiling = 100
T = 40

if T < melting:
    print("ice")
```

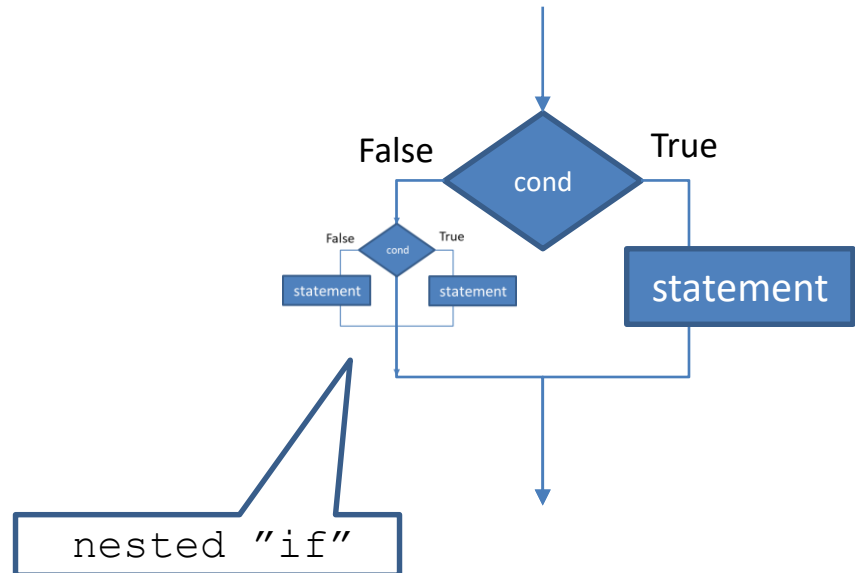


Conditional Statements

```
if T < melting:  
    print("ice")  
else:  
    print("liquid or gas")
```



```
if T < melting:  
    print("ice")  
else:  
    if T < boiling:  
        print("liquid")  
    else:  
        print("gas")
```



Conditional Statements

```
if T < melting:
    print("ice")
else:
    if T < boiling:
        print("liquid")
    else:
        print("gas")
```

==

```
if T < melting:
    print("ice")
elif T < boiling:
    print("liquid")
else:
    print("gas")
```

- We can have 0 or more `elif` blocks, `else` is optional
- What's the difference?

```
if T < melting:
    print("ice")
if T < boiling:
    print("liquid")
else:
    print("gas")
```

Important Syntactic Notes

- The **colon** (:) following the **if** statement acts as to open a new **scope** (similar to opening a parenthesis, or begin, in other programming languages).
- The **print** statement in the line below it is **indented one tab** to the right. This indicates it is within the scope of this **if**.
- The first statement not indented this way (in our case, the **else**) is **outside that scope**.
- In IDLE, you can use **tabs** or **spaces** for indentation. The width of the tab / number of spaces must be consistent within a single scope.
- Such consistency across the whole program is highly recommended as well. However sometimes inconsistency is practically unavoidable (can you think of such a scenario?)

Documenting Your Programs

- An essential part of writing computer code is **documenting** it.

```
melting = 0
boiling = 100
T = 40 # room temperature in Celsius

if T < melting:
    print("ice") # water below 0 Celsius
elif T < boiling:
    print("liquid") # water between 0 and 100 Celsius
else:
    print("gas") # water above 100 Celsius
```

Documenting Your Programs

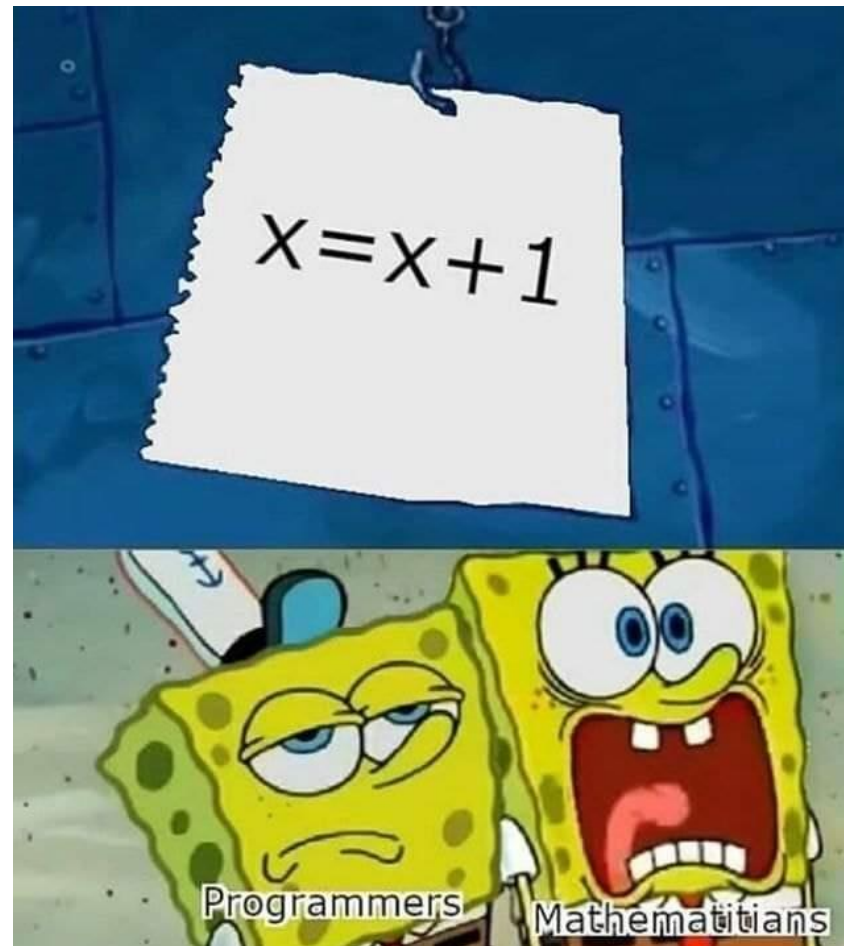
- An essential part of writing computer code is **documenting** it
 - for you to later understand what you had in mind
 - For your teammate to be able to coordinate her or his code with yours.
 - For the grader or teaching assistant in your class to try and understand your code, and grade it accordingly, etc., etc...
- To assist in documentation, all programming languages have **comments**. Comments are pieces of text that are **not interpreted** or executed by the computer.
- The simplest type of comments in Python are **one line comments**. They start with the hash character, **#**, and extend to the end of the physical line. Note that **#** within a string is treated as a character and not as a beginning of a comment.

```
# comments can start at beginning of a line
```

```
a=1 # comments can also start after the beginning of a line
```

```
"# but this is a string, NOT a comment"
```

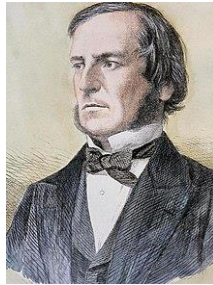
Comic Relief^{*}



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

Boolean Type

- **Boolean** values are either **true** or **false**.
- Note that Python's **True** and **False** are capitalized, while in most other programming languages they are not.



George Boole
1815–1864

- The standard **logical operators** **and**, **or**, **not** can be applied to them and generate complex Boolean expressions from “atomic” ones.

Boolean Type and Logical Operators

```
>>> a = True
>>> b = True
>>> c = False
```

```
>>> a and b
True
>>> a and c
False
>>> a or c
True
>>> a or False
True
>>> not a
False
```

Binary ops

| a | b | a and b | a or b |
|---|---|---------|--------|
| T | T | T | T |
| T | F | F | T |
| F | T | F | T |
| F | F | F | F |

| a | not a |
|---|-------|
| T | F |
| F | T |

Unary op

More on Boolean Type

- We could settle with only **and** and **not**:

not (a **or** b) is equivalent to (**not** a) **and** (**not** b)

- You may have seen this equivalence in the **Discrete Math** course (**De Morgan rules**):

$$\overline{A \vee B} \equiv \bar{A} \wedge \bar{B}$$

- Similarly, we could settle with **or** and **not**.
- In fact, either combination is **universal**, meaning that it is enough to represent any logical operator of 1 or more variables (you may prove this claim in the “computer structure” course)

Exclusive Or (XOR)

- Python does **not** have a built-in Boolean **xor** (exclusive or) operator, which is a highly useful operator:

| a | b | a xor b |
|---|---|-------------------------|
| T | T | F (note the difference) |
| T | F | T |
| F | T | T |
| F | F | F |

```
>>> a = True
>>> b = True
>>> c = False
```

```
>>> (a and (not b)) or ((not a) and b)
False
```

```
>>> (a and (not c)) or ((not a) and c)
True
```

a xor b

a xor c

Exclusive Or (XOR)

```
>>> (a and (not b)) or ((not a) and b)
```

```
False
```

```
>>> (a and (not c)) or ((not a) and c)
```

```
True
```

- It is annoying and time consuming to write and rewrite the same expression with different variables. We will address this issue when we discuss functions (in the next lecture). Then we will be able to write:

```
>>> xor(a, b)
```

```
False
```



Not a built-in Python command.
We will implement it soon

Comparison Operators

- Comparing numbers is important in many contexts. Python's comparison operators are intended for that: they operate on two numbers and return a **Boolean** value, indicating equality, inequality, or a size comparison.

```
>>> 5 == 5
```

```
True
```

```
>>> 6 != 6
```

```
False
```

```
>>> 3 > 2
```

```
True
```

```
>>> 4 < 3
```

```
False
```

```
>>> 3 >= 2
```

```
True
```

```
>>> 4 <= 4
```

```
True
```

6≠6

Shortcut for 3>2 or 3==2

Comparison Operators (cont.)

- We can compare numbers of **different types**. The interpreter implicitly **coerces** (casts) the more restricted type to the wider one, and performs the comparison.

```
>>> 19.1 > 7
```

```
True
```

```
>>> 14.0 == 14
```

```
???
```

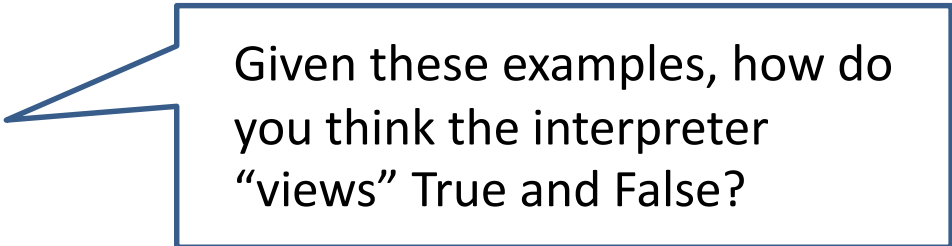


Check it! Don't be lazy!

Comparing Booleans

- What about comparing **Booleans**? Well, instead of guessing, let us try:

```
>>> True > True
False
>>> True > False
True
>>> False > 2.17
False
>>> 4 + True
5
>>> False * 2
0
>>> True + False
1
```



Given these examples, how do you think the interpreter “views” True and False?

- Python “views” True as 1 and False as 0.

```
>>> True == 1 and False == 0
True
```

- Yet, we strongly recommend you do not use True and False in arithmetic contexts

What Else Can We Compare?

```
>>> "0" == 0
```

```
False
```

```
>>> "0" > 0
```

```
Traceback (most recent call last):
```

```
File "<pyshell#19>", line 1, in <module>
```

```
"0">0
```

```
TypeError: unorderable types: str() > int()
```

- Fair enough.
- What about comparing strings to strings?

Comparing Strings

- What about comparing strings to strings?

```
>>> "Amir" >= "Amir"
```

```
True
```

```
>>> "Amir" > "Amir"
```

```
False
```

```
>>> "Michal" > "Amir"
```

```
True
```

```
>>> "Amir" > "Michal"
```

```
False
```

```
>>> "Amirrrrrrrrrrr" > "Michal"
```

```
False
```

```
>>> "cat" > "bat"
```

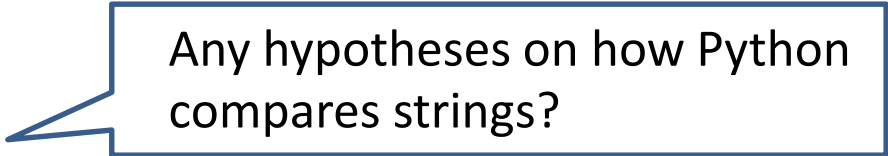
```
True
```

```
>>> "cat" > "cut"
```

```
False
```

```
>>> "cat" > "car"
```

```
True
```



Any hypotheses on how Python compares strings?

Lexicographical (alphabetical) order

- Assumption: the set of **alphabet** (characters allowed) is **totally ordered**. This means that any 2 single characters can be compared.
- This assumption holds in Python (more details in the future, but if you are curious, look [here](#))
- Given 2 strings over some totally ordered alphabet

$$S = s_0s_1 \cdots s_{n-1} \quad \text{and} \quad T = t_0t_1 \cdots t_{m-1}$$

$S < T$ if and only if

there **exists** some $i \geq 0$ such that

(1) for **every** $0 \leq j < i$ we have $s_j = t_j$ and

(2) either $s_i < t_i$ or $i = n < m$.

Comic Relief*

- What to do after class:



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

Loops and Iteration

*“**Iteration** means the act of **repeating** a process usually with the aim of **approaching** a desired goal or **target** or result. Each repetition of the process is also called an ‘iteration,’ and the results of one iteration are used as the **starting point for the next iteration**.”*

(from Wikipedia)

Our Example

- It is very common to have a portion of a program where we wish to iterate the same operation, possibly with different arguments. For example,

```
>>> 1+2+3+4+5+6+7+8+9+10  
55
```

- If we just want to add the numbers from 1 to 10, the piece of code above may be adequate. But suppose that instead, we wish to increase the summation limit to 100, 1000, or even 10^8 .
- We obviously ought to have a more **efficient** method to express such iterated addition. We want to iterate over the integers in the **range** [1,100], [1,1000], or [1, 10^8], repeatedly, adding the next element to the partially computed sum.

Wait a Minute...



from Wikipedia

- Do you know young Carl Friedrich Gauss (1777 – 1855)?
- At the age of six, he allegedly figured out how to **efficiently** compute this sum, which is an **arithmetic series**.
- Gauss' observation was that $1 + 2 + \dots + n = \frac{n(n+1)}{2}$.
- We are **not** going to use it, though, and simply let the computer chew its way through the iteration.

while Loops

```
n = 10**8
```

```
i = 1
```

```
s = 0
```

```
while i<=n:
```

```
    s = s+i
```

```
    i = i+1
```

```
    #print("i=", i, "s=", s)
```

```
print(s)
```

```
5000000050000000
```

loop
condition

loop
"body"

outside
loop

- The loop is entered and **executed as long as** the loop **condition** ($i \leq n$ in this case) is **True**.
- Notice the colon and indentation (**tabs**), which define the **scope** ("body") of the loop

while Loops Cauties

```
n = 10**8
```

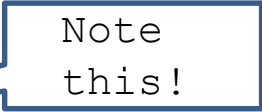
```
i = 1
```

```
s = 0
```

```
while i<=n:
```

```
    s = s+i
```

```
    i = i+1
```



Note
this!

```
        #print("i=", i, "s=", s)
```

```
print(s)
```

```
???
```

while Loops Cavities

```
n = 10**8
```

```
i = n
```

```
s = 0
```

```
while i<=n:
```

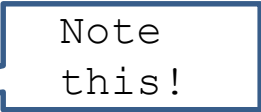
```
    s = s+i
```

```
    i = i+1
```

```
    #print("i=", i, "s=", s)
```

```
print(s)
```

```
???
```



Note
this!

breaking Loops

```
n = 10**8

i = 1
s = 0
while i<=n:
    s = s+i
    i = i+1
    if i==100:
        break
    #print("i=", i, "s=", s)
print(s)
```

- **break** **terminates** the nearest enclosing loop, skipping any other code that follows the break inside the loop. It may only occur syntactically in loops. Useful for getting out of loops when some predefined condition occurs.

continue-ing Loops

```
n = 10**8

i = 1
s = 0
while i<=n:
    s = s+i
    i = i+1
    if i==100:
        continue
    #print("i=", i, "s=", s)
print(s)
```

- Check it yourselves!

for Loops (on strings)

```
for char in "abcd":  
    print(char)
```

a
b
c
d

```
for char in "abcd":  
    print((char+"!")*2)  
    print(str.replace((char+"!")*2, "!", "?"))  
???
```

- `for` and `in` are reserved words of Python, causing iterated execution, where the variable `char` goes over all elements in `"abcd"`.
- Notice the `colon` and `indentation`, which determine the `scope` where the iteration occurs.

for Loops on range

- Python has a type `range`, which is an **ordered collection** of all **integers** in a given range.
- `range(n)` defines the sequence $0, 1, 2, \dots, n - 1$
- that is, all integers k satisfying $0 \leq k < n$

```
for num in range(10):
```

```
    print(num)
```

0

1

2

3

4

5

6

7

8

9

Python's `range`

- More generally, `range(a, b)` for two integers a, b , contains all integers k satisfying $a \leq k < b$.
 - Note that if $a > b$, `range(a, b)` is `empty` (however, this is not an error).
 - So `range(n)` is a shorthand for `range(0, n)`.
- Even more generally, `range(a, b, d)` contains all integers of the form $a + i \cdot d$, satisfying $a \leq a + i \cdot d < b$ ($i > 0$)
 - This is an `arithmetic progressions` (סדרה חשבונית)
 - So `range(a, b)` is a shorthand for `range(a, b, 1)`.
 - Note that for $d = 0$, `range(a, b, d)` results in an error.
 - Can we use $d < 0$? Try it!

for Loops – Back to Our Example

```
n = 10**8
```

```
s = 0
```

```
for i in range(1,n+1):
```

```
    s = s+i
```

```
print(s)
```

```
5000000050000000
```

Yet another Solution, using Python's Built-in `sum`

- Summation over a structure is so common that Python has a built-in function for it, `sum`, enabling an even more concise code.

```
>>> n = 10**8  
>>> sum(range(1,n+1))  
5000000050000000
```

Comparing Solution's Efficiency

- Young Gauss' observation enabled him to calculate the sum $1 + 2 + \dots + 100$ very fast.

$$1 + 2 + \dots + n = \frac{n(n + 1)}{2}$$

- In modern terminology, we could say that his method, requiring only **one addition**, **one multiplication** and **one division**, is much more computationally **efficient** than our method, requiring exactly **$n - 1$ addition operations**.
- We will define these notions in a precise manner in one of the next lectures, on **complexity**.

Comparing Solution's Efficiency

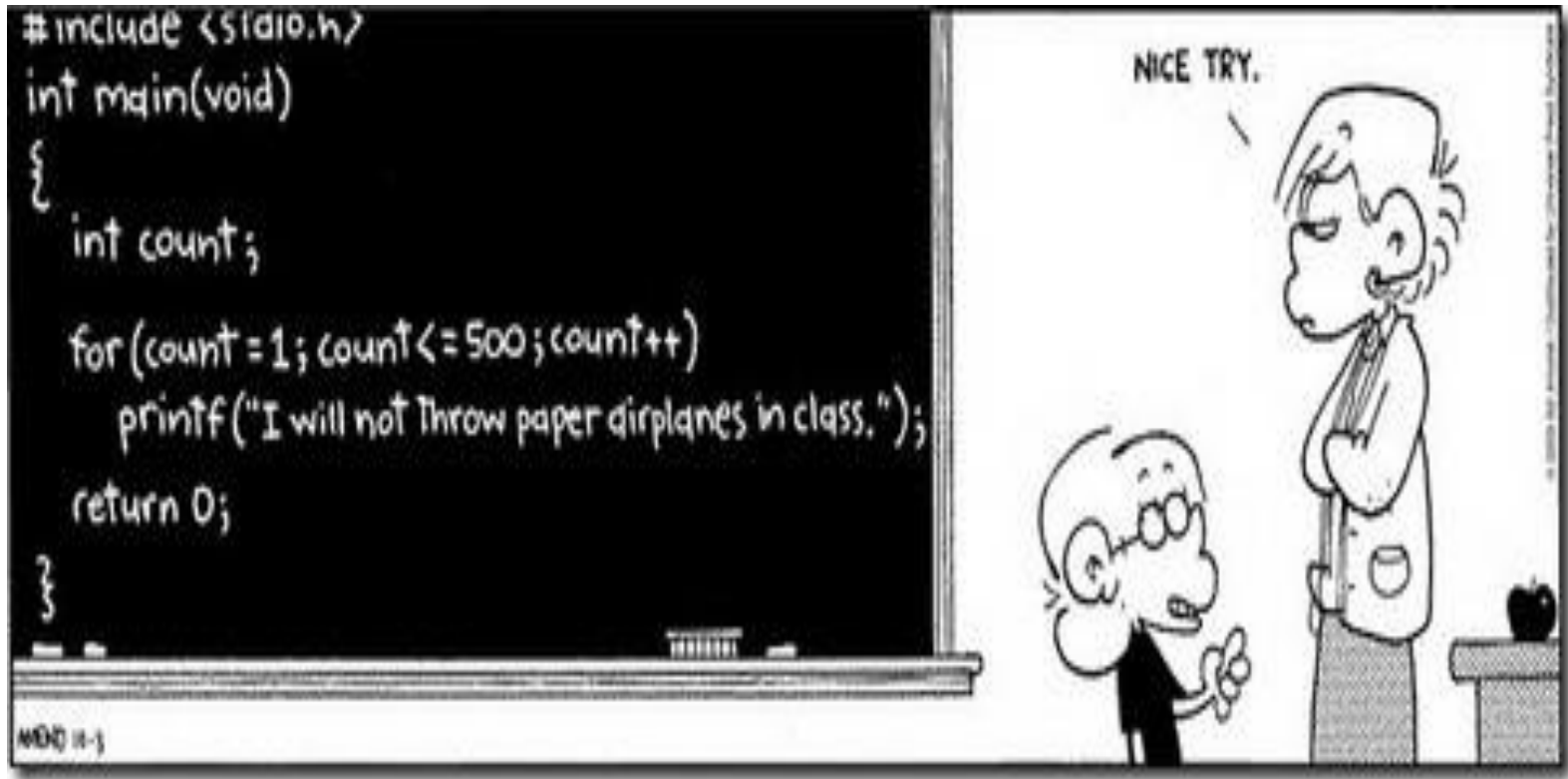
```
>>> n = 10**8  
>>> sum(range(1, n+1))  
5000000050000000
```

- Computing this sum takes almost **10^8 additions**. Even without measuring it, the time taken for the computation to complete is noticeable.

```
>>> n*(n+1)//2  
5000000050000000
```

- Computing this sum “the Gauss way” requires only **3 arithmetic operations** and is noticeably faster.
- Good algorithms often succeed to tackle problems in non-obvious ways, and dramatically improve their efficiency.

Comic Relief*

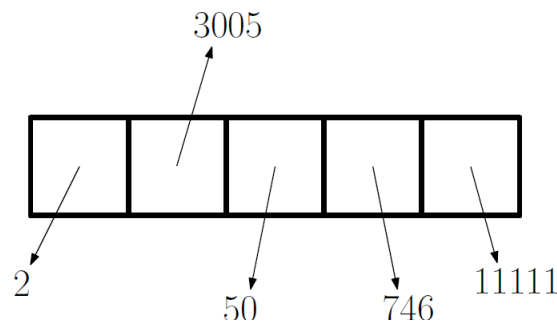


* אני מזמין אתכם לשלוח לי הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Type list

- `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

- 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
```

len()

- Quite often, the length is a useful quantity to have.
- `len` returns the **length** (number of elements) of a collection

```
>>> len("abcd")
```

```
4
```

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

```
>>> len(my_list)
```

```
5
```

```
>>> len([])
```

```
0
```

```
>>> len([1,2,3] + ["a", "b"])
```

```
5
```

List concatenation

- Collections in Python **store** and maintain their length, so applying `len()` to them involves merely a single **memory read** operation, thus is efficient

Iterating over Lists and Strings

- Recall that `for` loops allow simple iteration over collections:

```
for x in collection:  
    do something with x
```

```
for ch in "abc":  
    print(ch)
```

a
b
c

```
for x in [55, -6, 7, 8]:  
    print(x)
```

55
-6
7
8

Iteration using `range`

`for x in sequence:`
do something with **x**



```
L = [11, 12, 13]
```

```
for k in L:  
    print(k)
```

`for i in range(len(sequence)):`
do something with **sequence[i]**



```
L = [11, 12, 13]
```

```
for i in range(len(L)):  
    print(L[i])
```

Slicing Lists and Strings

- **Slicing** allows creating a **new** list (or string) from an existing one, where the new list (string) is composed of an **ordered subset** of the original one
- Slicing merely provides a **convenient shorthand** for a loop
- Slicing has many **parameters and options**. You should **neither** be overwhelmed by this, **nor** are you expected to digest and memorize all options right away. You **should** know the options exist and how to look them up.

Slicing – examples

```
>>> num_list = [11,12,13,14,15,16,17,18,19,20]
```

```
>>> len(num_list)
```

```
10
```

```
>>> num_list[1:5]          slicing
```

```
[12,13,14,15]
```

```
>>> num_list[0:10:2]      slicing an arithmetic progression
```

```
[11,13,15,17,19]
```

```
>>> num_list[::2]         shorthand for previous slicing
```

```
[11,13,15,17,19]
```

```
>>> num_list[::-1]        reversing the list
```

```
[20,19,18,17,16,15,14,13,12,11]
```

Slicing - examples (cont.)

```
>>> num_list[10::-1]          same as before  
[20,19,18,17,16,15,14,13,12,11]
```

```
>>> num_list[-1::-1]         index -1 refers to last element  
[20,19,18,17,16,15,14,13,12,11]
```

```
>>> num_list[-1:-11:-1]      and -11 here is one before first  
[20,19,18,17,16,15,14,13,12,11]
```

```
>>> num_list[8:3:-2]         arithmetic progression with  $\delta = -2$   
[19,17,15]
```

```
>>> num_list[3:8:-2]         outcome is an empty list, NOT an error  
[]
```

```
>>> num_list                 slicing did NOT change original list  
[11,12,13,14,15,16,17,18,19,20]
```

They Slice **Strings** Too, Don't They?

```
>>> len("Rye Bread")
```

```
9
```

```
>>> "Rye Bread"[0:9]
```

everything

```
'Rye Bread'
```

```
>>> "Rye Bread"[:]
```

shorthand for previous

```
'Rye Bread'
```

```
>>> "Rye Bread"[:4]
```

first 4 characters

```
'Rye '
```

```
>>> "Rye Bread"[4:]
```

everything but first 4 characters

```
'Bread'
```

```
>>> "Rye Bread"[:4] + "Rye Bread"[4:]
```

concatenate prefix+suffix

```
'Rye Bread'
```

Sliced Strings (cont.)

```
>>> "Rye Bread"[0:9:2]    every second char, starting from first
'ReBed'

>>> "Rye Bread"[0::2]    shorthand for previous
'ReBed'

>>> "Rye Bread"[:9:2]    shorthand for previous previous
'ReBed'

>>> "Rye Bread"[::2]    shorthand for previous previous previous
'ReBed'

>>> "Rye Bread"[9::-1]    everything, backwards
'daerB eyR'

>>> "Rye Bread"[::-1]    shorthand for previous
'daerB eyR'
```

Lists and Strings – Summary

- Both lists and strings are examples for **sequences** (ordered collections) in python
- Both can be **indexed, sliced** and **iterated over**
- More collections (ordered and unordered) coming soon

Lecture 2 - Highlights

- **Conditional statements** (`if-elif-else`) allow splitting the program's flow into paths
- Don't forget to **#document** your code
- Boolean values are either `True` or `False`
- **Logical operators**: `and`, `or`, `not` .
- **Comparison operators**: `==` `!=` `<` `>` `<=` `>=`
- Logical and comparison **expressions** evaluate to **Boolean** values
- In Python strings are compared **lexicographically**
- **Loops** allow repeating a set of operations multiple times ("**iterations**")
- `while` loops - as long as some condition holds
- `for` loops - over a given collection of elements (e.g., list, string, range)
- Python's **list** is an array of elements
 - Similar to strings, lists support indexing, iteration, slicing, `len()`