Intro2cs

Tirgul 4

What we will cover today?

- File IO
- Mutable and Immutable
- Aliasing
- Shallow and Deep copy
- local and global variables
- List comprehension
- Indexing and Slicing - short recap

File IO

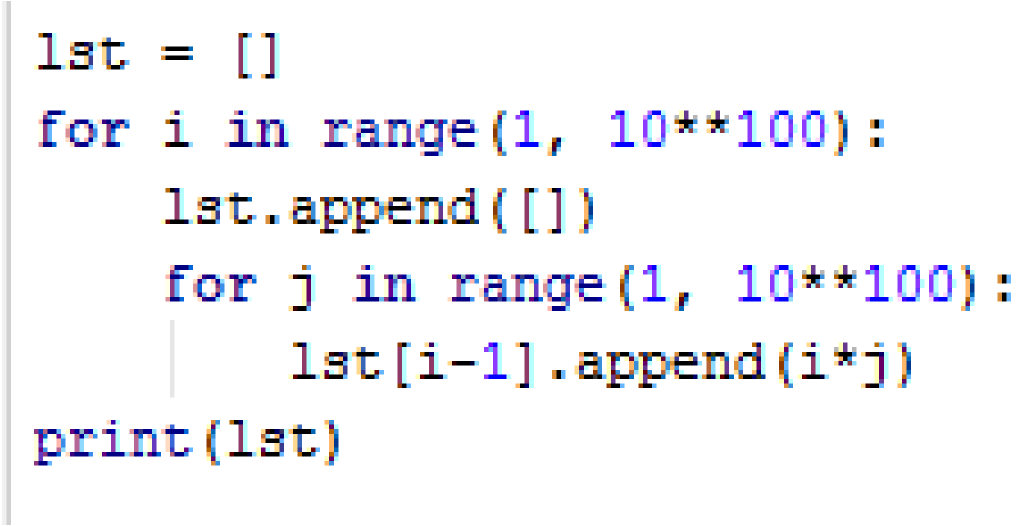
How Can We Store Data

- Till now, when we wanted to keep any piece of data we assigned to it variables of different types – integers, strings, lists, tuples etc.

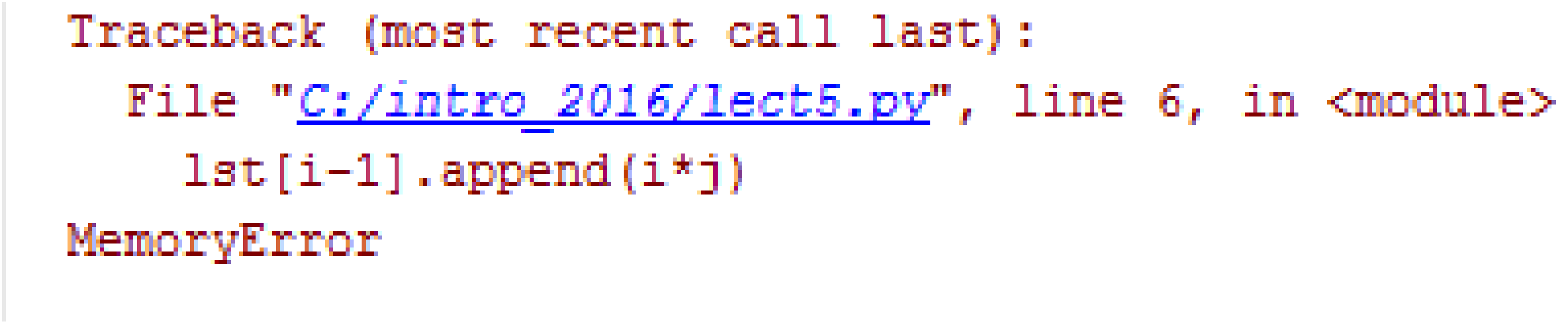
- Is there any limitation to the size of the data that can be assigned to a variable?

How Large is the Data We Use?

What will happen here?



```
lst = []
for i in range(1, 10**100):
    lst.append([])
    for j in range(1, 10**100):
        lst[i-1].append(i*j)
print(lst)
```

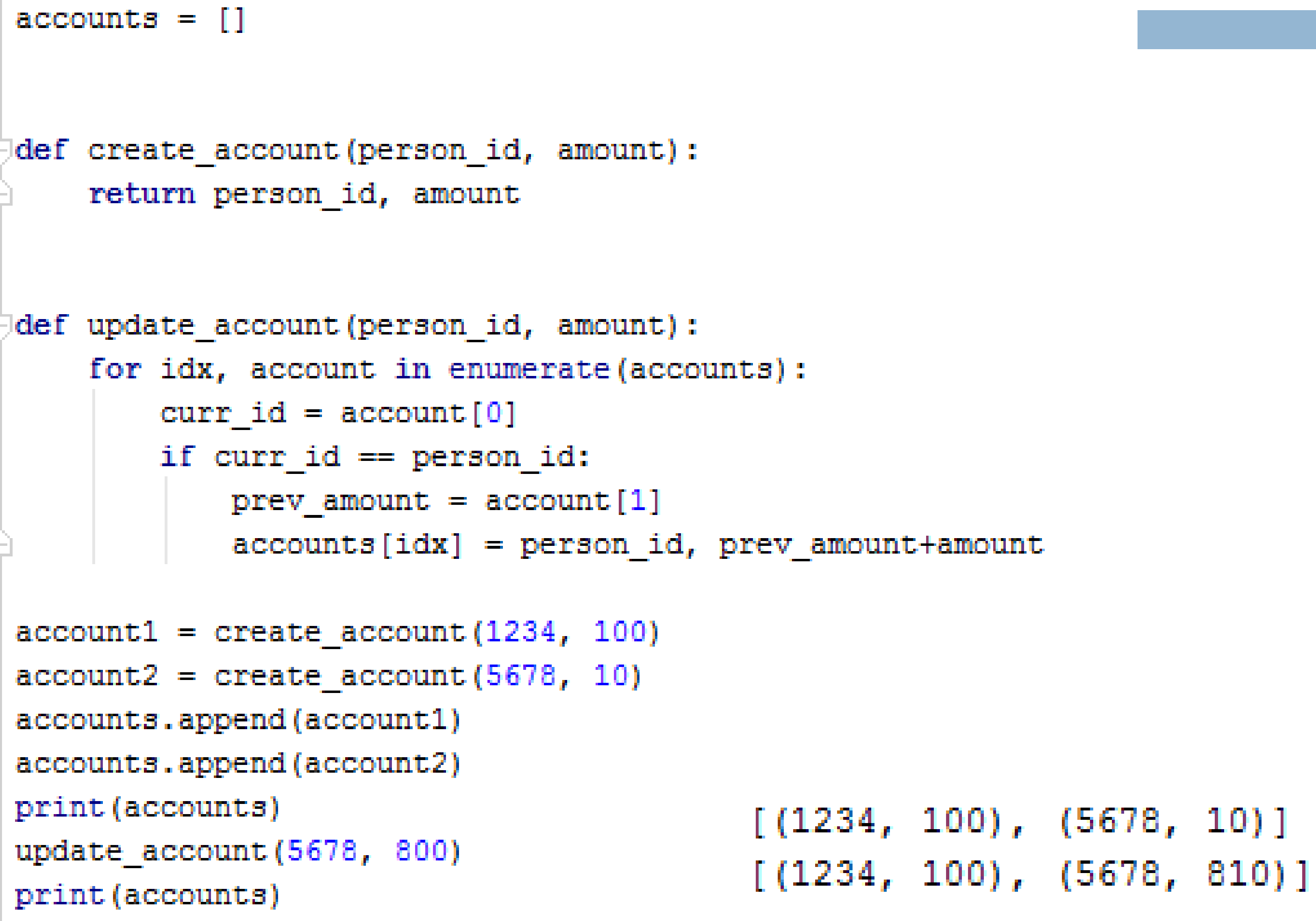


```
Traceback (most recent call last):
  File "C:/intro_2016/lect5.py", line 6, in <module>
    lst[i-1].append(i*j)
MemoryError
```

Memory Limitation

- The containers we know (list\tuple etc.) are limited in the amount of information they can hold (~536M items on a 32bit system[ref])

How Long is the Data Kept?



```
accounts = []


def create_account(person_id, amount):
    return person_id, amount


def update_account(person_id, amount):
    for idx, account in enumerate(accounts):
        curr_id = account[0]
        if curr_id == person_id:
            prev_amount = account[1]
            accounts[idx] = person_id, prev_amount+amount

account1 = create_account(1234, 100)
account2 = create_account(5678, 10)
accounts.append(account1)
accounts.append(account2)
print(accounts)
update_account(5678, 800)
print(accounts)
```

```
[(1234, 100), (5678, 10)]
[(1234, 100), (5678, 810)]
```

How Long is the Data Kept?

But what happens to the accounts data when the program finishes to run, for any reason?

That data is lost, since it is kept only while the program is running

Solution – we can use files!

❑A file is an ordered sequence of bytes.

❑Can store large amount of data

❑Files information is maintained across system shut downs\reboots, and could be accessed by different programs.

❑Upon reading a file, a suitable program interprets the content of a file (playing music, displaying images and text).

Handling Files in Python

To create files, write to files and read from files **Python defines a custom file handling API** (Application Programming Interface).

Creating a file object

```
f = open(filename)
```

- `open` is python built-in function which returns a file object, with which we can work with files.

- `filename` is a string which indicates the **file location** within the file system (absolute or relative to the folder from which the program is running).

  For example:

  C:/intro/ex5/some_txt_file.txt

File ‘open’ing mode

```
f = open(filename,[mode])
```

- When opening a file we should **declare our intentional** use of this file.
  - Protects us against undesired consequences.

- Two basic types of modes :
  - Read (default)
  - Write

File ‘open’ing mode (2)

- The multiple modes “defend” us so we would not do what we don’t mean to (e.g unwanted changes in the file)

- We have to declare in advance what are our intentions toward the open file (do we wish to manipulate it? read only?)

open in read mode (default)

```
f = open(filename,'r')
```

- If **filename** does not exist :

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    f = open('filename')
FileNotFoundError: [Errno 2] No such file or directory:
'filename'
```

open in write mode

```
f = open(filename,'w')
```

- If `filename` does not exist:

  python will create such

- If `filename` does exist:

  python will **override** (delete) the existing version and replace it with a new (the current) file. – Beware!

open in append mode

```
f = open(filename,'a')
```

- If `filename` does not exist:
  python will create such.

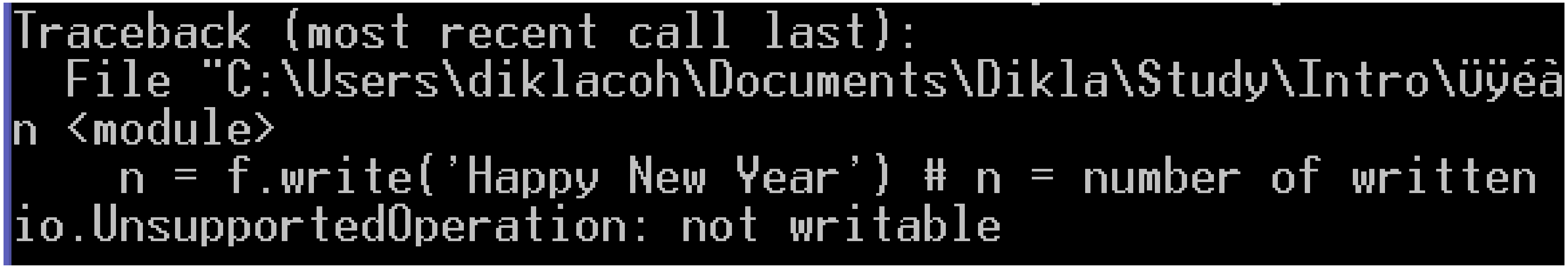
- If `filename` does exist:
  python will **add** any written data
  to the end of the file.

write to file

`n = f.write(s)`

- ❑ Write the string **`s`** to the file **`f`**.
- ❑ **`n`** = number of written characters.

```
f = open("file.txt")
n = f.write('Happy New Year')
```



```
Traceback (most recent call last):
  File "C:\Users\diklacoh\Documents\Dikla\Study\Intro\üÿéà
n <module>
    n = f.write('Happy New Year') # n = number of written
io.UnsupportedOperation: not writable
```

write to file (2)

```
f = open("file.txt", 'a')
n = f.write('Happy New Year')
```

- `write` adds content to the end of the file but does not insert line breaks.

- E.g :

```
f.write('Don')
f.write('key')
```

`f` content :

Happy New YearDonkey

write multiple lines – use Join

```
f = open("file.txt", 'a')
L = [‘Happy’,’new’,’year’]
n = f.write('\n'.join(L))
```

`f` content :

```
 Happy
 new
 year
```

- **Join - Concatenate strings using the first string**

write multiple terms to file

`f.writelines(seq)`

- ❑ **`writelines`** Write the strings contained in **`seq`** to the file one by one.
- ❑ For example:

```
f = open("file.txt", 'a')
my_strings = ['Don', ‘key']
f.writelines(my_strings)
```

`f` content :

```
Donkey
```

- ❑ **`writelines`** expects a list of strings, while **`write`** expects a single string.

closeing a file

```
f.close()
```

❑ After completing the usage of the file, it should be closed using the `close` method.

- Free the file resource for usage of other processes.

- Some environments will not allow read-only and write modes opening simultaneously

the open with statement

- Things don’t always go as smoothly as we plan, and sometimes causes programs to crash.
    - E.g. trying a 0 division.

- If the program crashes, we don’t get a chance to close (and free the resource of) the open file.

- To verify the appropriate handling of files, even in case of program crash we can make use of the **with** statement.

the with statement

```
with open(file_path, 'w') as our_open_file:
    # we can work here with the open
    # file and be sure it is properly
    # closed in every scenario
```

- The with statement guarantees that even if the program crashed inside the **with** block, it will still be properly closed.

You can’t write to a closed file

```
f.close()
f.writelines(['hi','bi','pi'])
```

```
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    f.writelines(['hi','bi','pi'])
ValueError: I/O operation on closed file.
```

reading from file

`f.read(n)`

- ❑ Read at most **n** characters from **`f`** (default = all file)

`f.readline()`

- ❑Read **one** line.

`f.readlines()`

- ❑Read **all lines** in the file - returns a list of strings (list of the file’s lines).

https://docs.python.org/3/tutorial/inputoutput.html

reading a 'w' mode file

```
f = open(filename,'w')
f.read()
```

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    f.read()
io.UnsupportedOperation: not readable
```

- Use ‘**r+**’ mode for **reading and writing**

Iterating over a file

When we read a file, Python defines a pointer (an *iterator*) which advances with every consecutive reading.

```
f.readline() # 1st time
 >> Hi :)
f.readline() # 2nd time
 >> Bye :(
```

f content

Hi :)
Bye :(
Mitz
Paz

Iterating over a file - tell

The `tell()` method returns the current position within the file.

```
f.readline() # 1st time
 >> Hi :)
f.tell()
 >> 5
```

f content

```
Hi :)
Bye :(
Mitz
Paz
```

Access (e.g. print) all the lines in a file

```
f = open(filename)
for line in f:
    print(line)
```

❑ When do the iteration stop?
When reaching end-of-file (EOF)

❑ We may think on a file as a sequence - thus we stop when there’s no more items in f - when we reach EOF

Programs with multiple files

Importing files

Suppose `foo.py` has code we want to use in `bar.py` .
In `bar.py` we can write :

```
import foo
foo.func1()
```

```
import foo as my_foo
my_foo.func1()
```

preferred alternatives. No problems with name collision

```
from foo import func1,func2
func1()
```

```
from foo import *
func1()
```

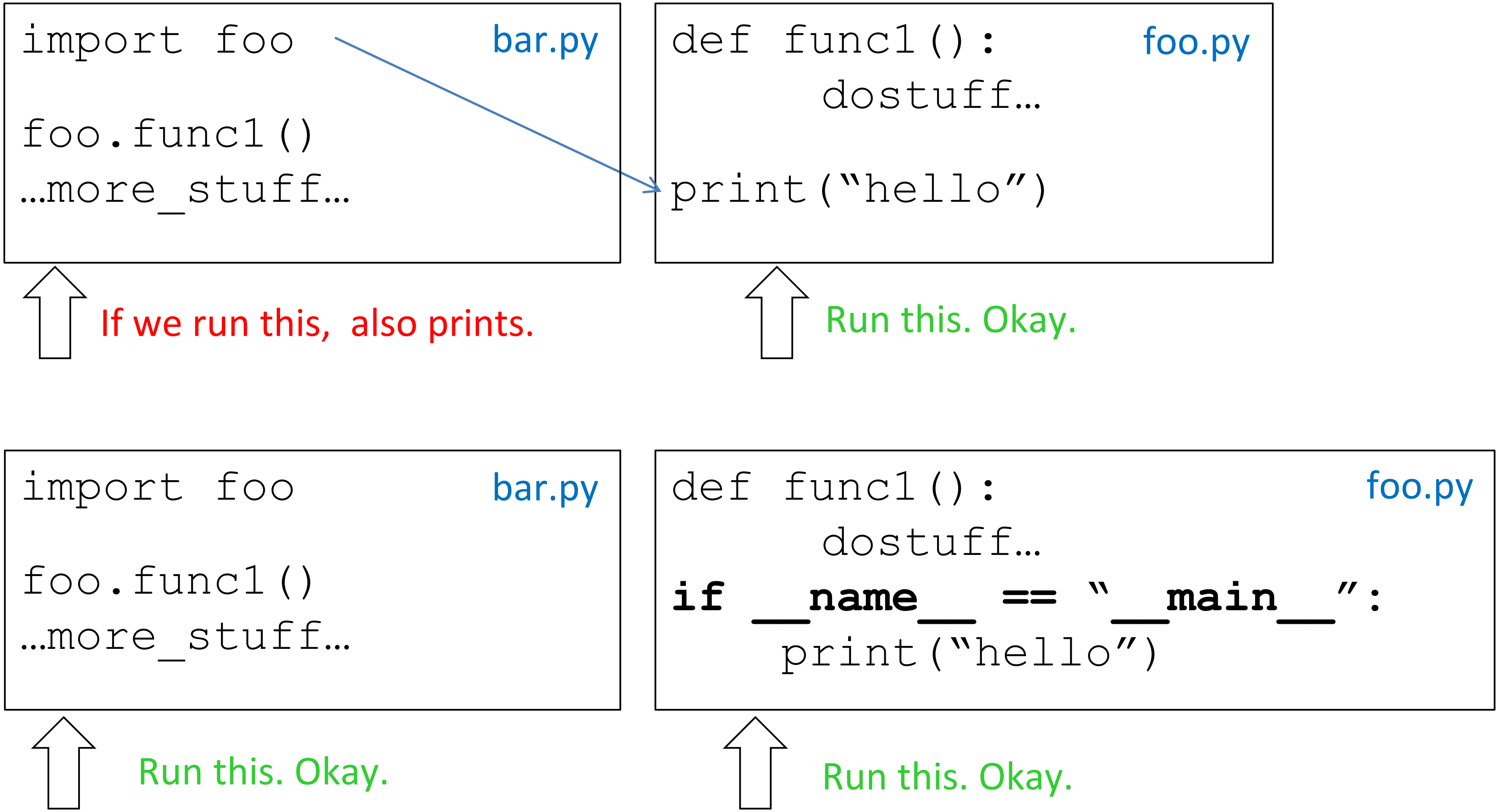
Here’s more info on handling cyclic imports. https://docs.python.org/3/faq/programming.html#how-can-i-have-modules-that-mutually-import-each-other

if __name__ == “__main__”:

- Sometimes we want to write a .py file that can be both used by other programs and/or modules as a module, and can also be run as the main program itself.
- __name__ will have the value “__main__” only if the current module is first to execute.

Importing runs the code

We want to be able to run foo.py, but also want to use its code in bar.py



bar.py

```
import foo

foo.func1()
…more_stuff…
```

If we run this, also prints.

foo.py

```
def func1():
        dostuff…

print("hello")
```

Run this. Okay.

bar.py

```
import foo

foo.func1()
…more_stuff…
```

Run this. Okay.

foo.py

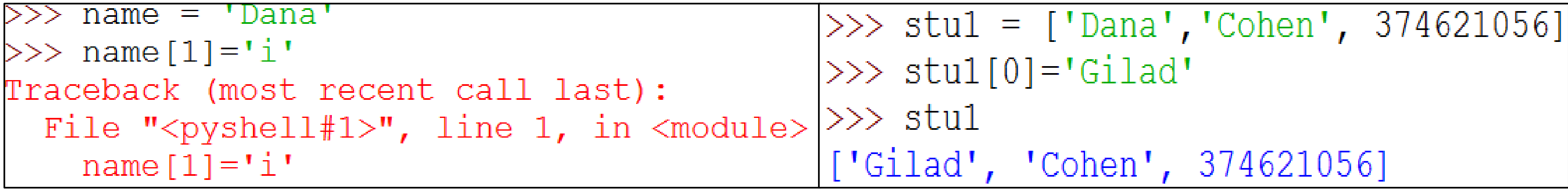
```
def func1():
        dostuff…
if __name__ == "__main__":
      print("hello")
```

Run this. Okay.

Mutable and Immutable

Mutable and Immutable

- Objects whose value can be changed are said to be *mutable* ;
- An object’s mutability is determined by its type: for instance, **numbers, strings, ranges** and **tuples** are immutable, while **lists** are mutable.



```
>>> name = 'Dana'
>>> name[1]='i'
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    name[1]='i'
```

```
>>> stu1 = ['Dana','Cohen', 374621056]
>>> stu1[0]='Gilad'
>>> stu1
['Gilad', 'Cohen', 374621056]
```

Immutability - Advantages

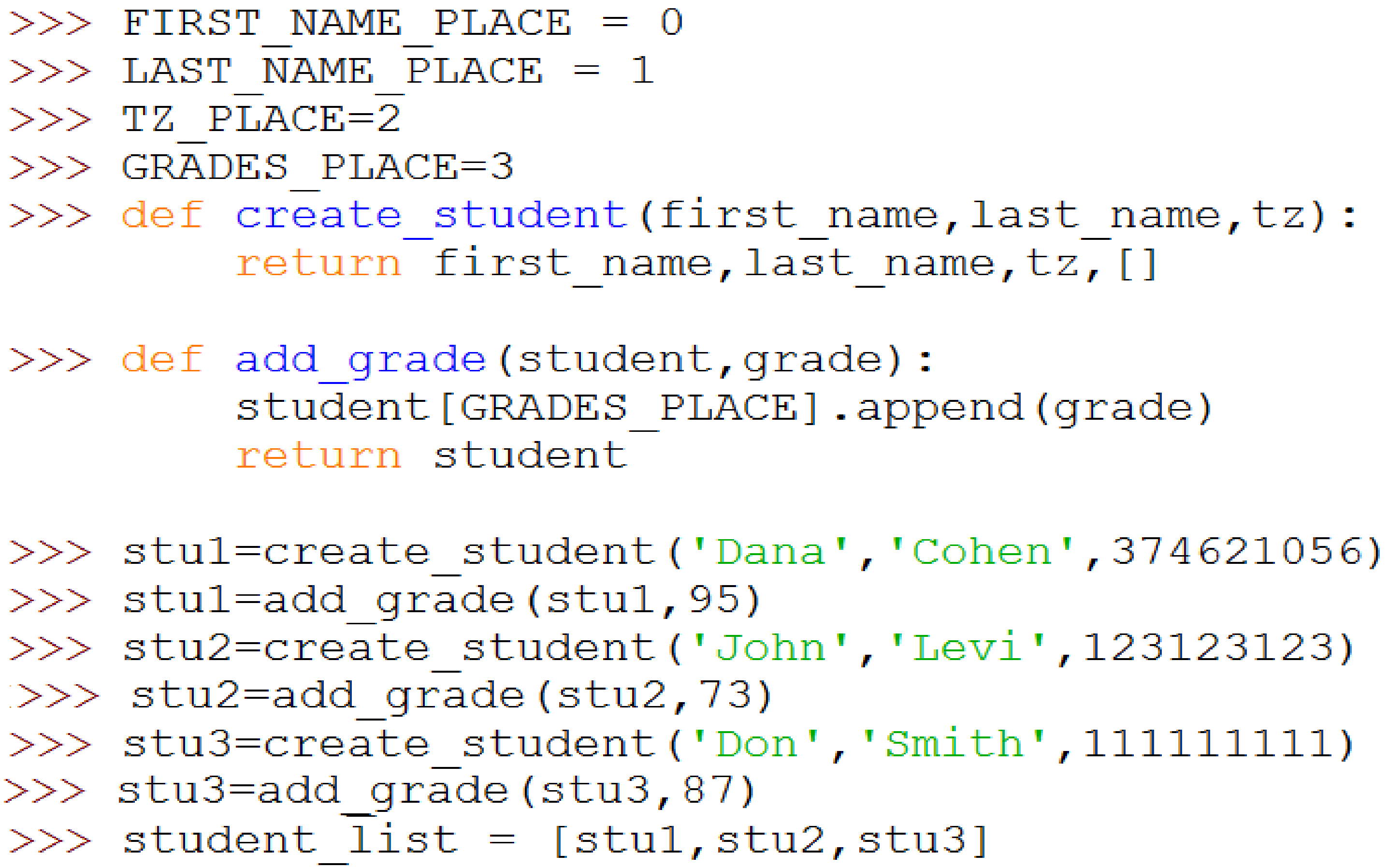
1. **Safe access**- allows others to use the object without changing the object

2. **Performance** – if the immutable objects are used to find other objects (dictionaries etc.) the interpreter will take advantage of the immutability – i.e., it will skip some calculations

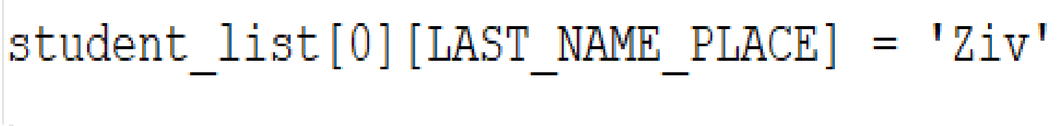
Mutable and Immutable types

Object	Mutable or Immutable?
int, float (a = 1, a = 1.5)	Immutable
string (a = “hello”)	Immutable
range (a = range(10))	Immutable
tuple (a = (1,2))	Immutable
list (a = [1,2])	Mutable

Immutable/Mutable Example



```
>>> FIRST_NAME_PLACE = 0
>>> LAST_NAME_PLACE = 1
>>> TZ_PLACE=2
>>> GRADES_PLACE=3
>>> def create_student(first_name,last_name,tz):
        return first_name,last_name,tz,[]

>>> def add_grade(student,grade):
        student[GRADES_PLACE].append(grade)
        return student

>>> stu1=create_student('Dana','Cohen',374621056)
>>> stu1=add_grade(stu1,95)
>>> stu2=create_student('John','Levi',123123123)
>>> stu2=add_grade(stu2,73)
>>> stu3=create_student('Don','Smith',111111111)
>>> stu3=add_grade(stu3,87)
>>> student_list = [stu1,stu2,stu3]
```



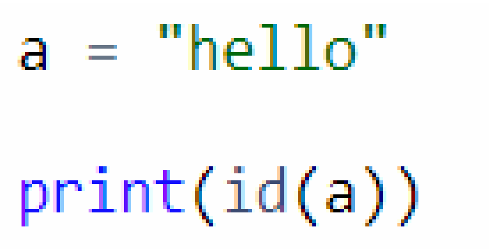
```
student_list[0][LAST_NAME_PLACE] = 'Ziv'
```

✗

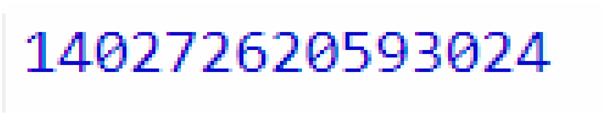
id function

id(object): returns the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime.

In most computers the id function return the address of the object in memory.



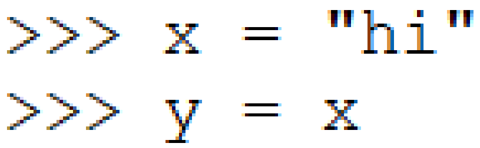
```
a = "hello"

print(id(a))
```



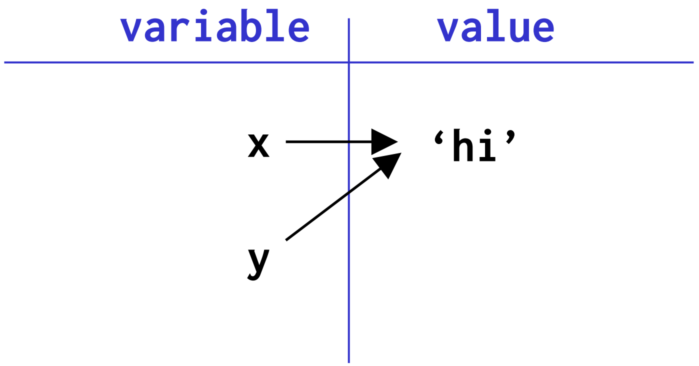
```
140272620593024
```

Aliases

- An *alias* is another name for a piece of data.
- Often easier (and more useful) than making a second copy.
- Aliasing happens whenever one variable's value is assigned to another variable using the = sign.

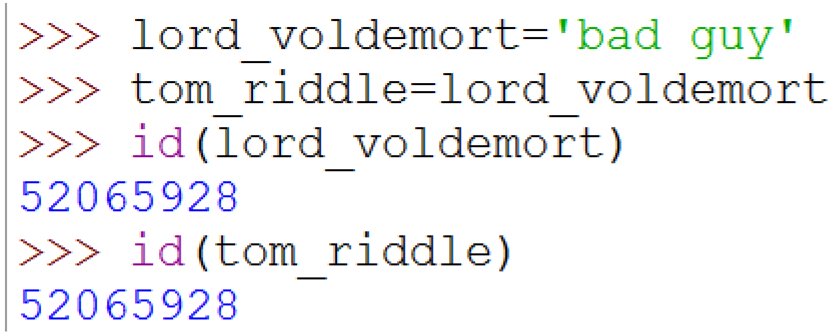


```
>>> x = "hi"
>>> y = x
```

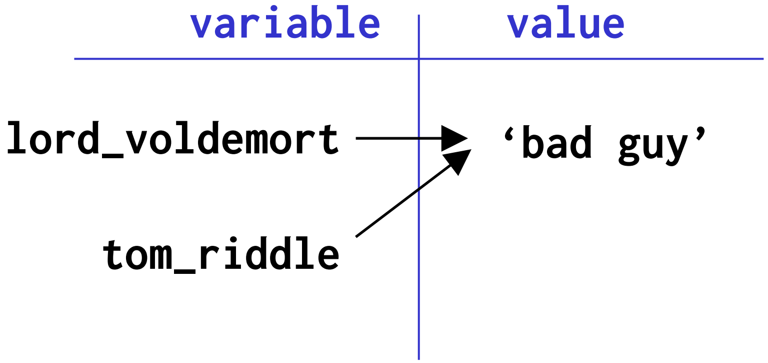


Aliasing immutable items – same identity

When we copy immutable items, the two items have the same id:



```
>>> lord_voldemort='bad guy'
>>> tom_riddle=lord_voldemort
>>> id(lord_voldemort)
52065928
>>> id(tom_riddle)
52065928
```



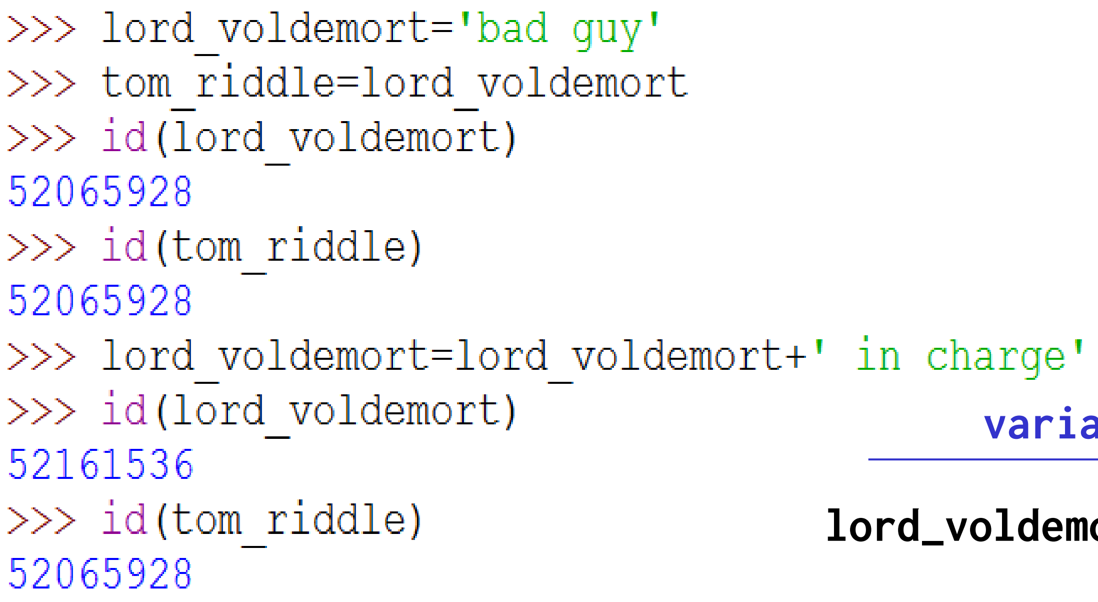
variable | value

lord_voldemort → ‘bad guy’

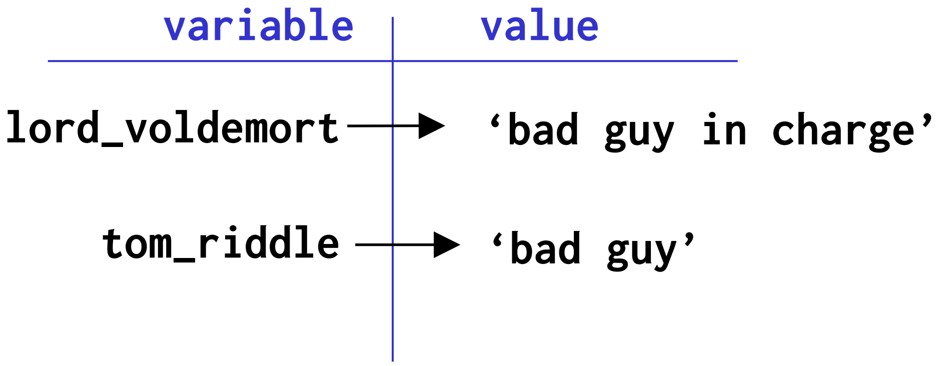
tom_riddle

Any new assignment will result in a different identity

But a new assignment will always result in a new identity:



```
>>> lord_voldemort='bad guy'
>>> tom_riddle=lord_voldemort
>>> id(lord_voldemort)
52065928
>>> id(tom_riddle)
52065928
>>> lord_voldemort=lord_voldemort+' in charge'
>>> id(lord_voldemort)
52161536
>>> id(tom_riddle)
52065928
```



variable value

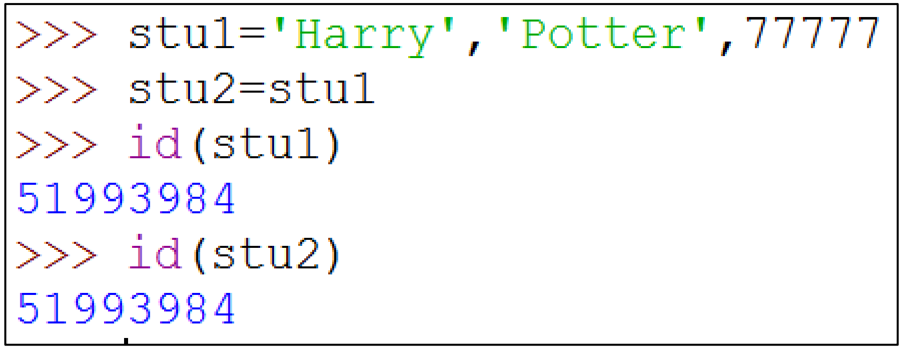
lord_voldemort → ‘bad guy in charge’

tom_riddle → ‘bad guy’

Copying with the = sign: copying a reference

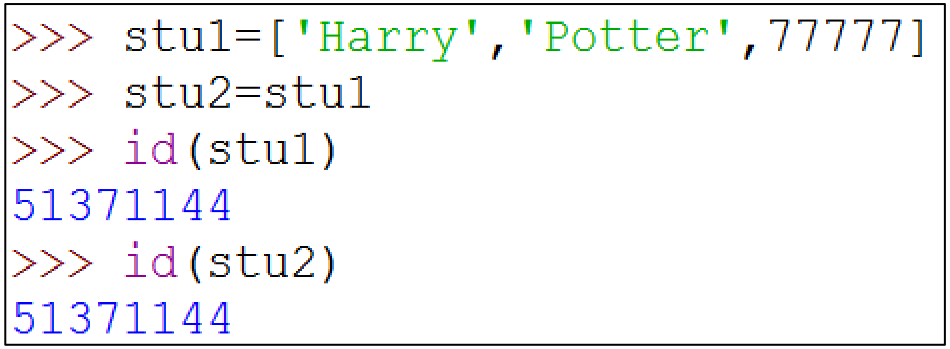
Assign a list to another - both point to the same place in memory! (like immutable objects)

tuple

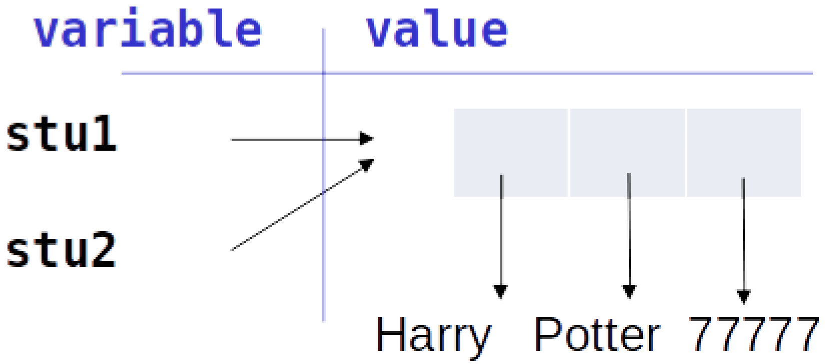


```
>>> stu1='Harry','Potter',77777
>>> stu2=stu1
>>> id(stu1)
51993984
>>> id(stu2)
51993984
```

list



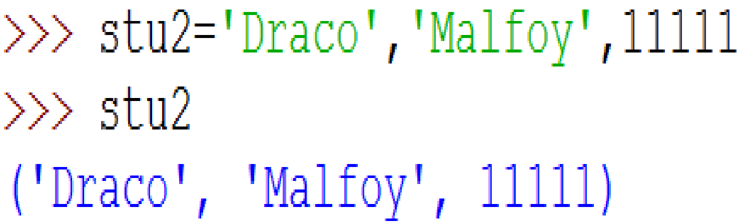
```
>>> stu1=['Harry','Potter',77777]
>>> stu2=stu1
>>> id(stu1)
51371144
>>> id(stu2)
51371144
```



Aliasing with the = operator

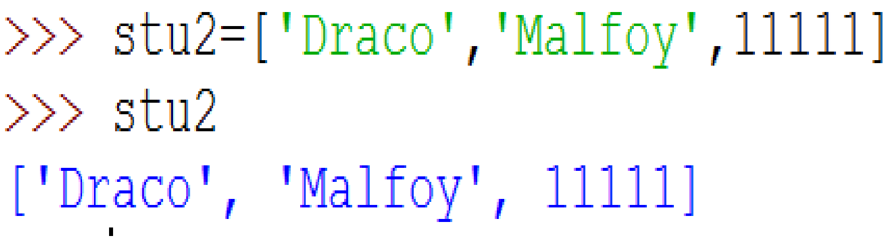
any new assignment will always lead to a new identity (like immutable objects)

tuple

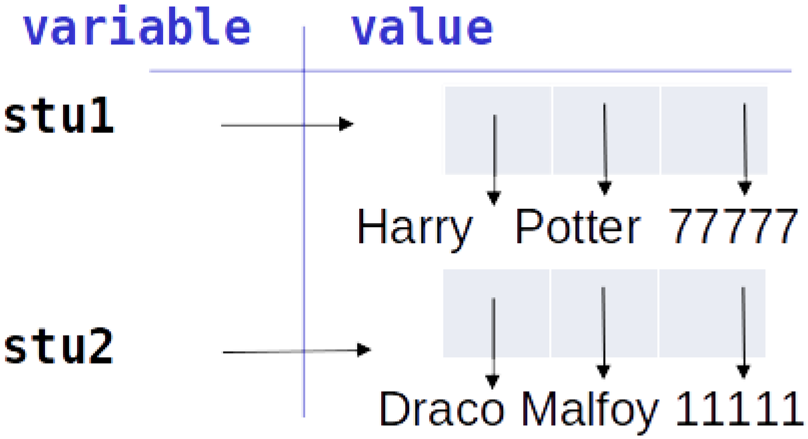


```
>>> stu2='Draco','Malfoy',11111
>>> stu2
('Draco', 'Malfoy', 11111)
```

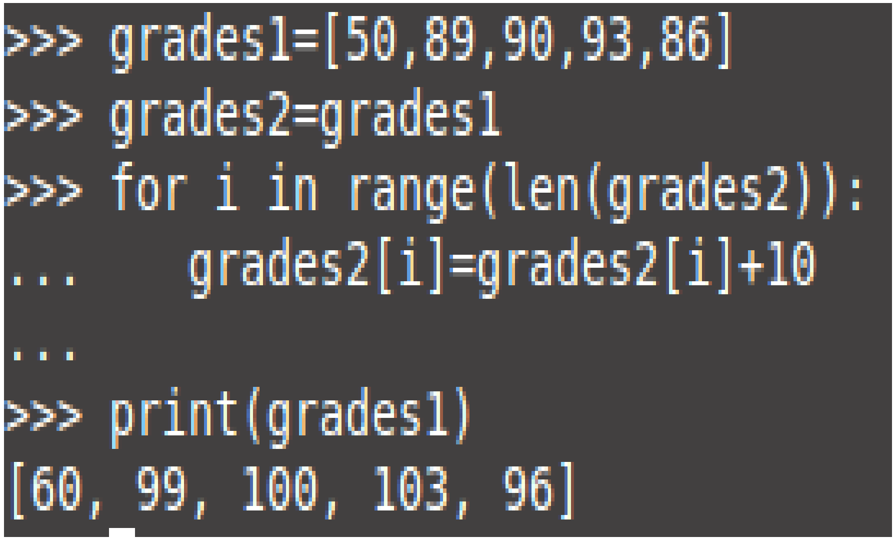
list



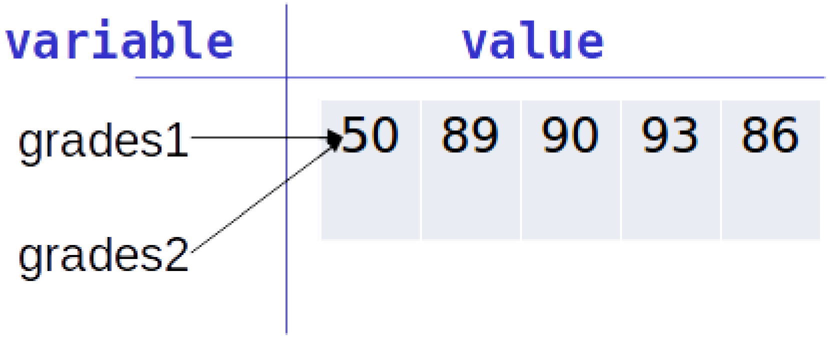
```
>>> stu2=['Draco','Malfoy',11111]
>>> stu2
['Draco', 'Malfoy', 11111]
```

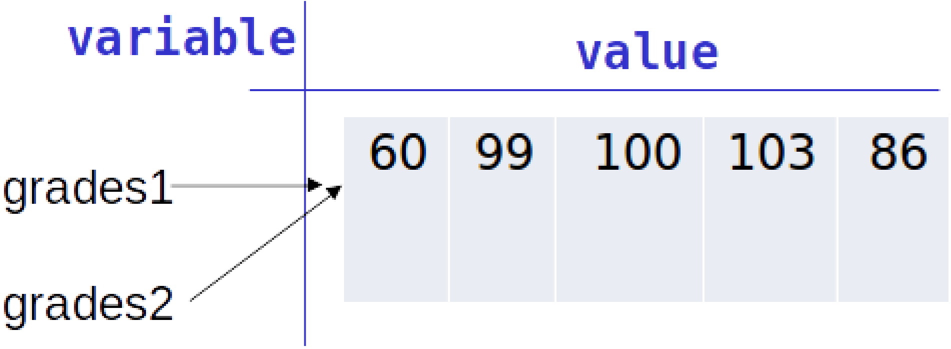


Aliasing with the = operator

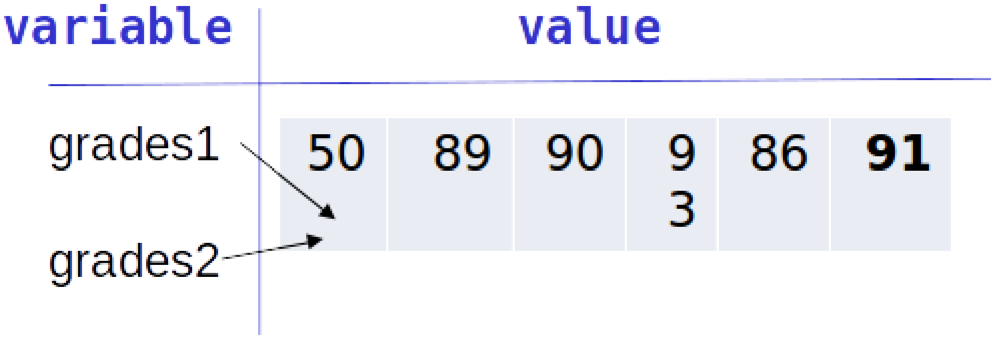


```
>>> grades1=[50,89,90,93,86]
>>> grades2=grades1
>>> for i in range(len(grades2)):
...     grades2[i]=grades2[i]+10
...
>>> print(grades1)
[60, 99, 100, 103, 96]
```





Aliasing with the = operator



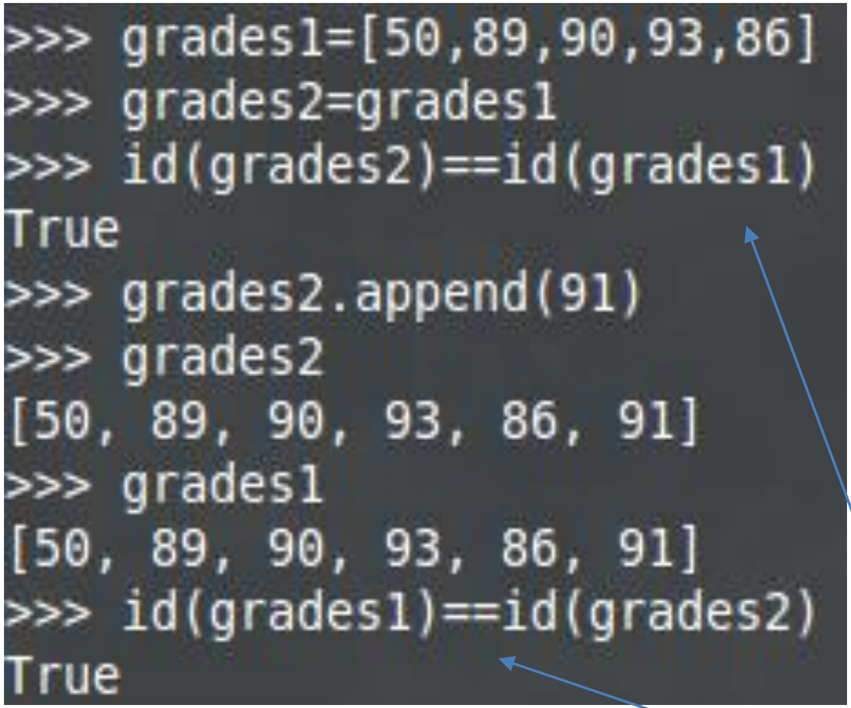
variable

value

grades1

grades2

50 89 90 93 86 **91**



```
>>> grades1=[50,89,90,93,86]
>>> grades2=grades1
>>> id(grades2)==id(grades1)
True
>>> grades2.append(91)
>>> grades2
[50, 89, 90, 93, 86, 91]
>>> grades1
[50, 89, 90, 93, 86, 91]
>>> id(grades1)==id(grades2)
True
```

Same Identity!

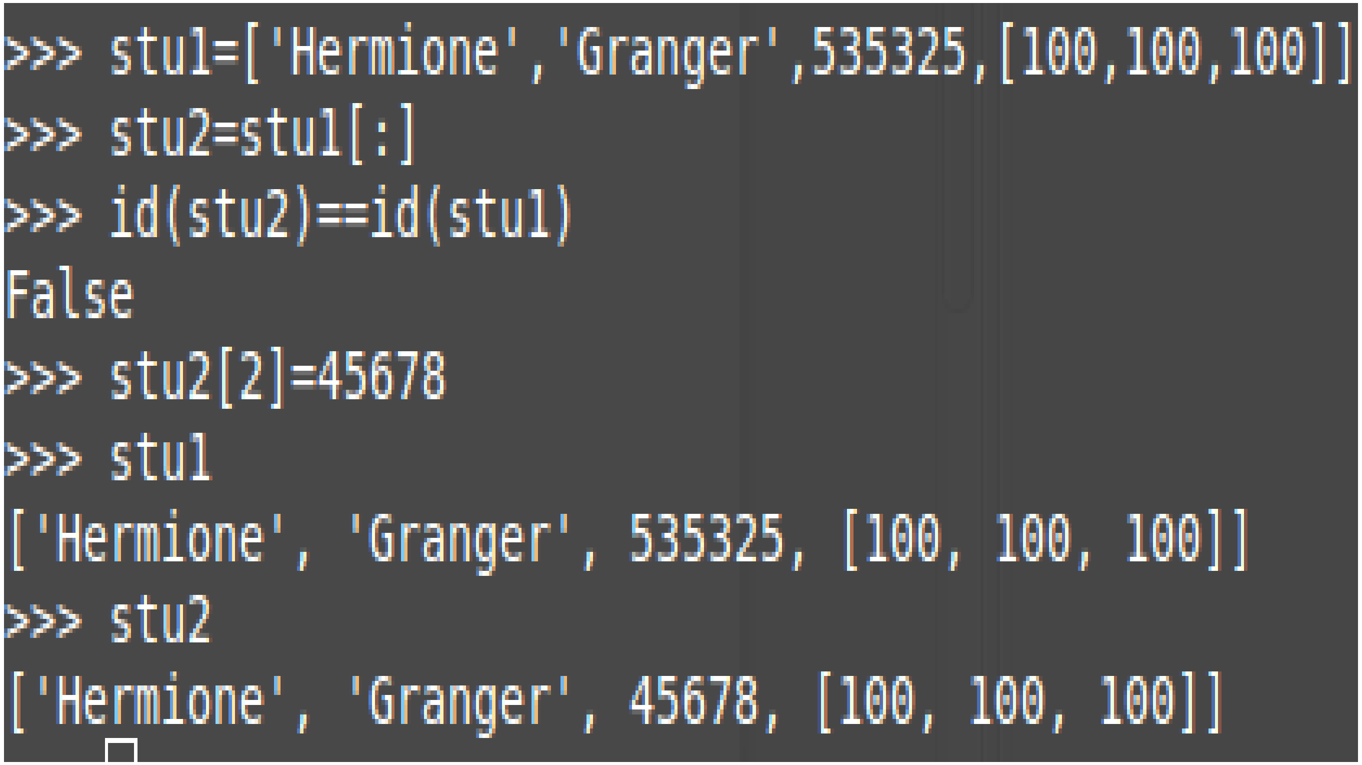
Shallow and Deep Copy

Shallow copy
In the process of shallow copying A, B will copy all of A's field values. If the field value is a memory address it copies the memory address, and if the field value is an object that is not containers: int, float, bool (primitive type) - it copies the value of it.

Deep copy
In deep copy the **data is actually copied over.** The result is different from the result a shallow copy gives. The advantage is that **A and B do not depend on each other,** but at the cost of a slower and more expensive copy.

Copy with the slice operator

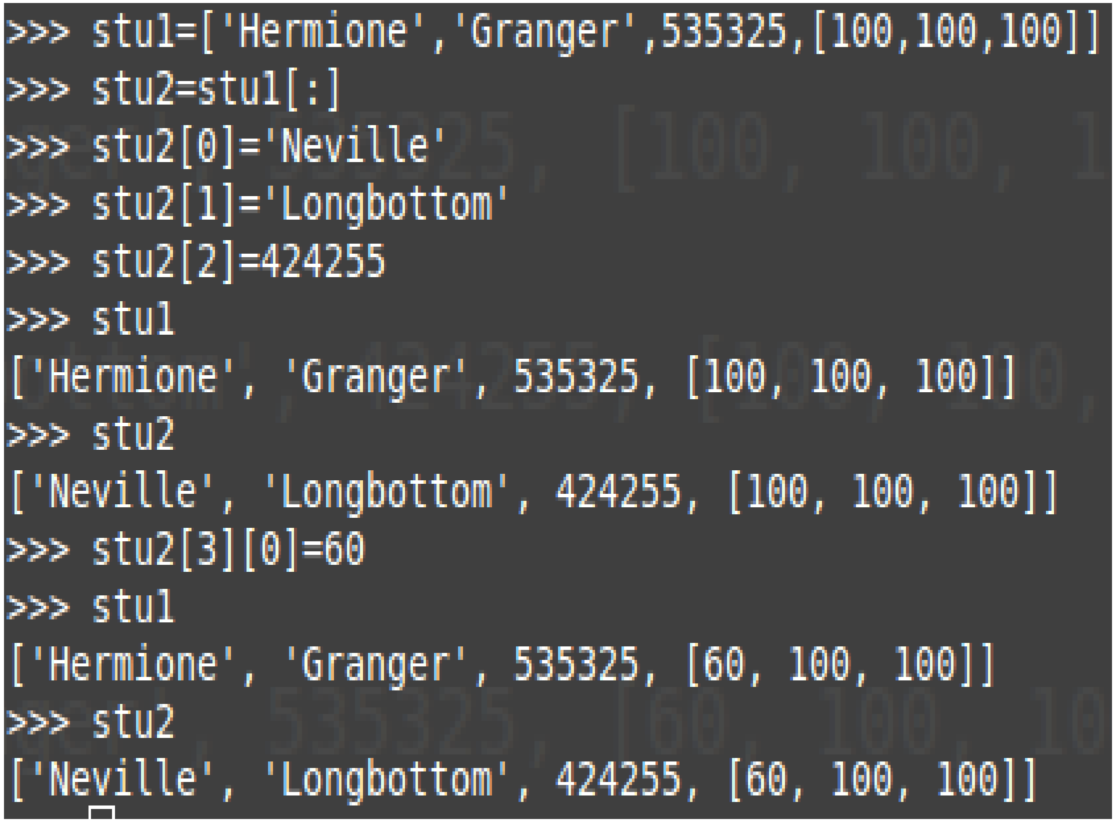
Simple data types



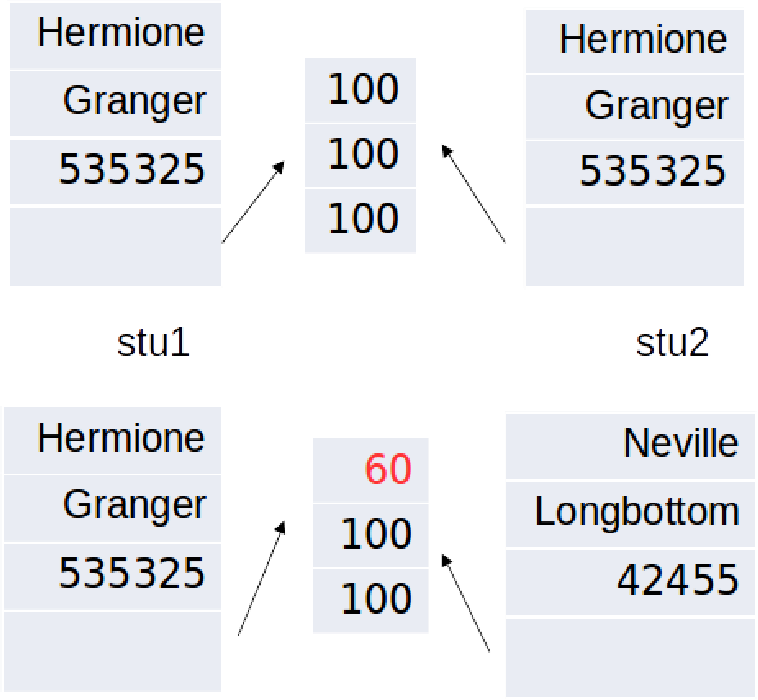
```
>>> stu1=['Hermione','Granger',535325,[100,100,100]]
>>> stu2=stu1[:]
>>> id(stu2)==id(stu1)
False
>>> stu2[2]=45678
>>> stu1
['Hermione', 'Granger', 535325, [100, 100, 100]]
>>> stu2
['Hermione', 'Granger', 45678, [100, 100, 100]]
```

Copy with the slice operator – shallow copy

Compound data types - copy the reference!

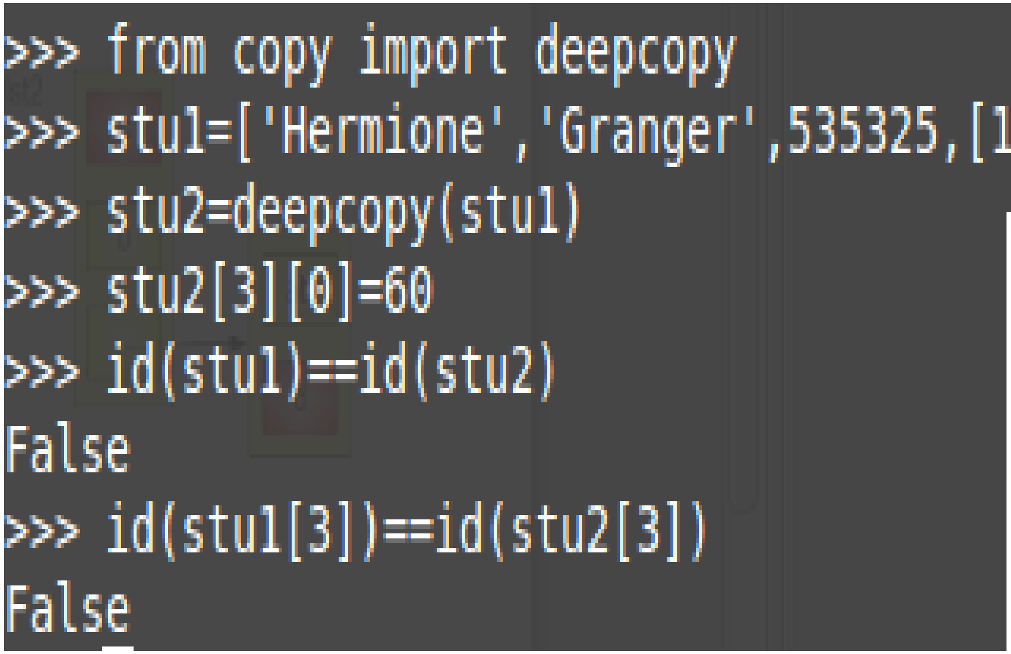


```
>>> stu1=['Hermione','Granger',535325,[100,100,100]]
>>> stu2=stu1[:]
>>> stu2[0]='Neville'
>>> stu2[1]='Longbottom'
>>> stu2[2]=424255
>>> stu1
['Hermione', 'Granger', 535325, [100, 100, 100]]
>>> stu2
['Neville', 'Longbottom', 424255, [100, 100, 100]]
>>> stu2[3][0]=60
>>> stu1
['Hermione', 'Granger', 535325, [60, 100, 100]]
>>> stu2
['Neville', 'Longbottom', 424255, [60, 100, 100]]
```

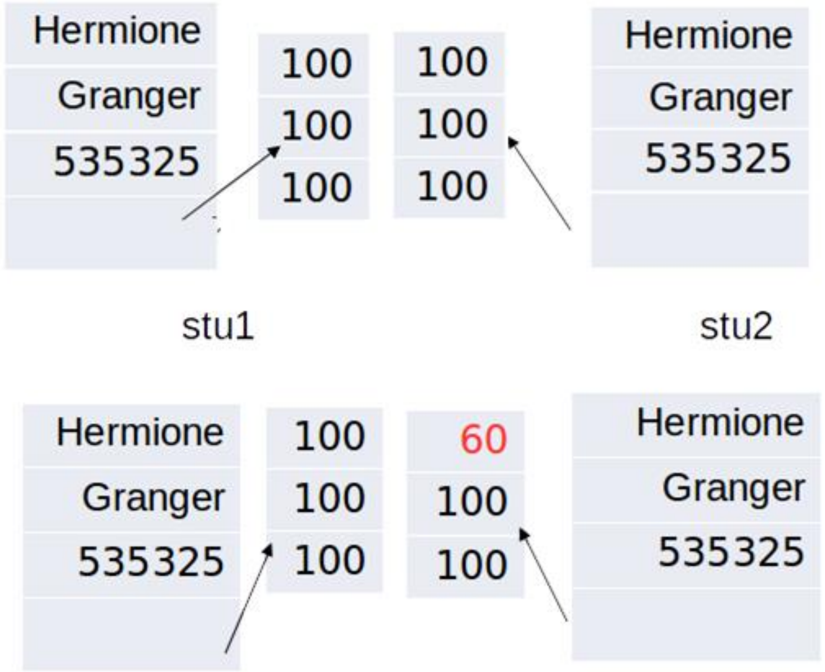


Deep copy from the Module copy

Create a new version of the same values but with new references!



```
>>> from copy import deepcopy
>>> stu1=['Hermione','Granger',535325,[100,100,100]]
>>> stu2=deepcopy(stu1)
>>> stu2[3][0]=60
>>> id(stu1)==id(stu2)
False
>>> id(stu1[3])==id(stu2[3])
False
```



Quick Summary - Mutable items

- In mutable items we distinguish between three cases:
- Assignment (=):

  Assign a second name to the same identity.
- Shallow copy ([:]):

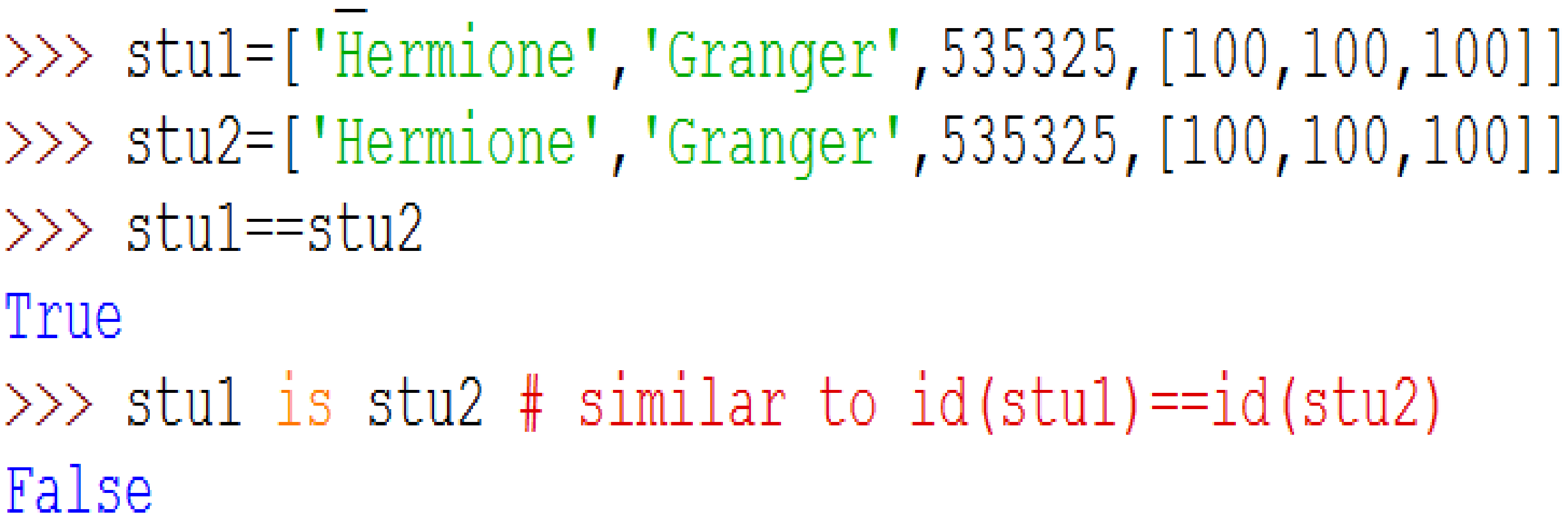
  Copy all the fields (values for simple types and references for memory addresses).
- Deep copy (copy.deepcopy)

  Copy (recursively) all the values in one list to another

is Vs. ==

- In Python we have two functions to compare objects: is and the == operator.
- is - tests for identity (references to the same object). a is b is just like writing id(a) == id(b).
- Equal (==) is a function that can be implemented differently (wait for the cool stuff in OOP).
- In Lists == works similar to deepcopy and compares the values of two lists

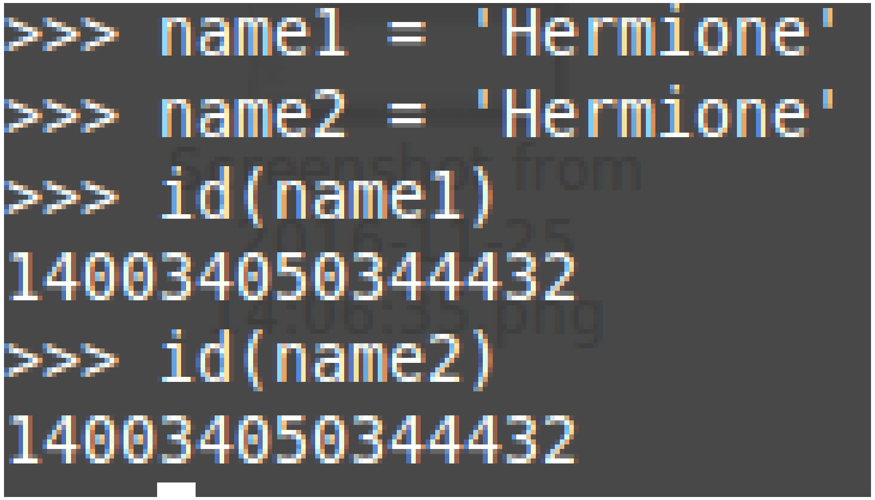
is vs. ==   example(1)



```
>>> stu1=['Hermione','Granger',535325,[100,100,100]]
>>> stu2=['Hermione','Granger',535325,[100,100,100]]
>>> stu1==stu2
True
>>> stu1 is stu2 # similar to id(stu1)==id(stu2)
False
```

is vs. ==   example(2)

- Sometimes, simple immutable values are shared – leading to the same ids!



```
>>> name1 = 'Hermione'
>>> name2 = 'Hermione'
>>> id(name1)
140034050344432
>>> id(name2)
140034050344432
```

Function scopes

- How can we determine what belongs to a function and what to the module? **Scopes**
- **Function Scope – Everything between the function definition and the end of the function.**
- Python functions have no explicit begin or end. The only delimiter is a colon (:) and the indentation of the code itself.
- **Everything inside the scope of a function "belongs" to it – local variables, local functions and more**

Local variables

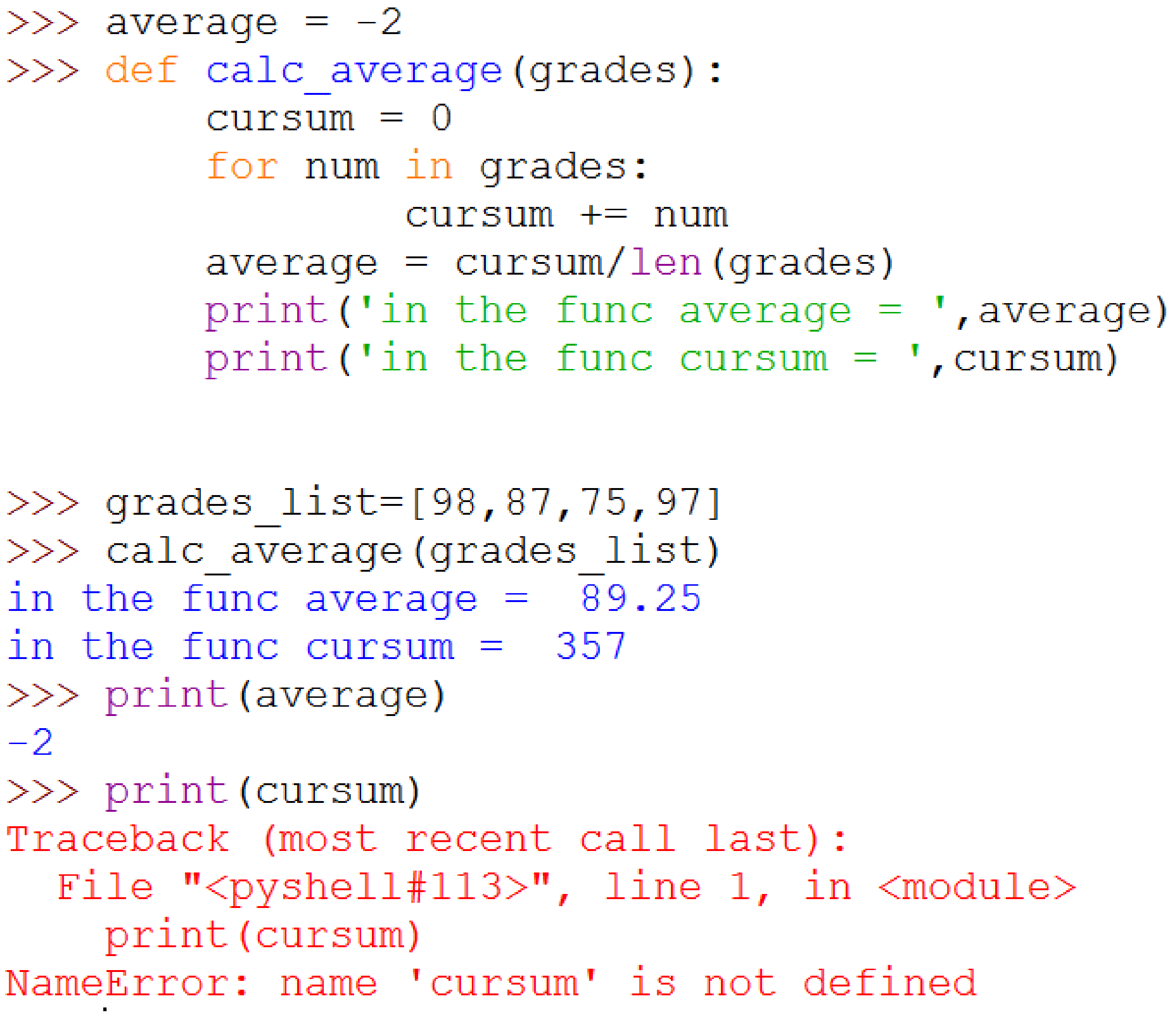
- Functions have a special type of variable called **local variables**
- **These variables only exist while the function is running.**
- Local variables **are not accessible from outside** the function.
- When a local variable has the same name as another variable (such as a global variable), the local variable hides the other.

Local variables – example (1)

Reminder:

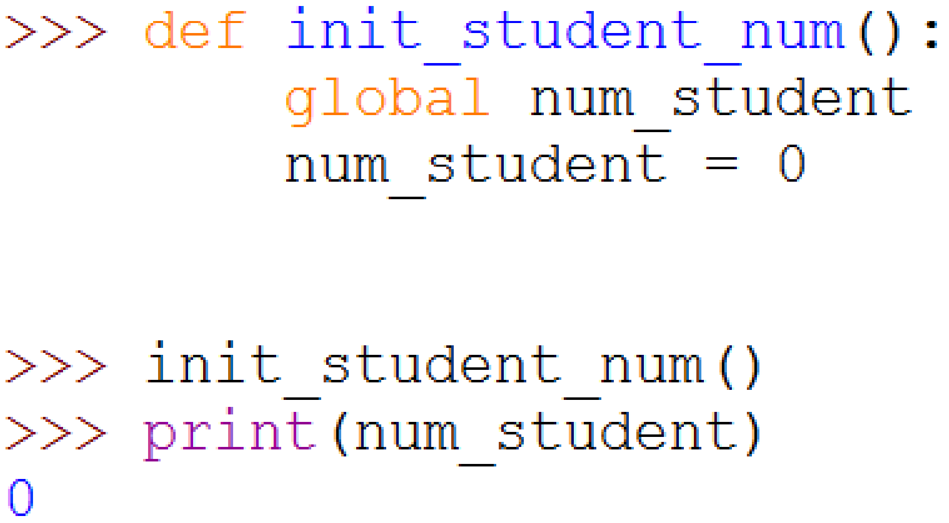
```
>>> def func_a():
          ...
          a = 3

>>> def func_b():
          ...
          b = a -2

>>> func_b()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func_b
NameError: name 'a' is not defined
```

Local variables – example(2)



```
>>> average = -2
>>> def calc_average(grades):
        cursum = 0
        for num in grades:
                cursum += num
        average = cursum/len(grades)
        print('in the func average = ',average)
        print('in the func cursum = ',cursum)


>>> grades_list=[98,87,75,97]
>>> calc_average(grades_list)
in the func average =  89.25
in the func cursum =  357
>>> print(average)
-2
>>> print(cursum)
Traceback (most recent call last):
  File "<pyshell#113>", line 1, in <module>
    print(cursum)
NameError: name 'cursum' is not defined
```

The `global` variable



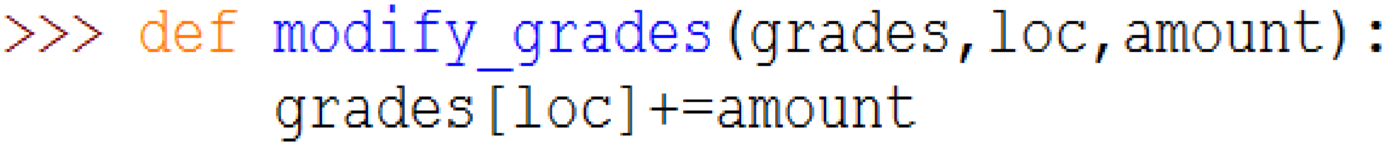
```
>>> def init_student_num():
        global num_student
        num_student = 0


>>> init_student_num()
>>> print(num_student)
0
```

The global variable

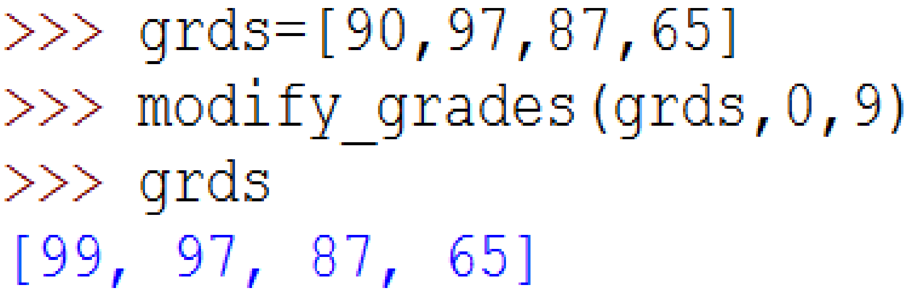
- When possible, avoid the use of global variables – global variable can potentially be modified from anywhere, and any part of the program may depend on it.

- Only use a global variable when it is used as a constant in multiple parts of the program

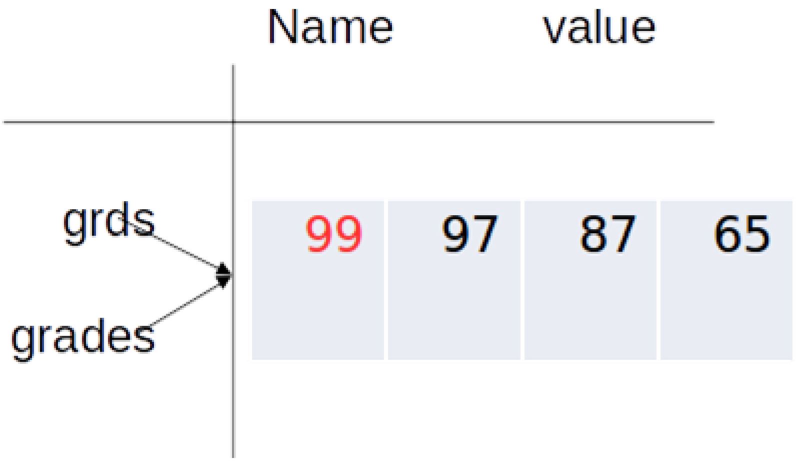
Mutable items as variables



```
>>> def modify_grades(grades,loc,amount):
        grades[loc]+=amount
```



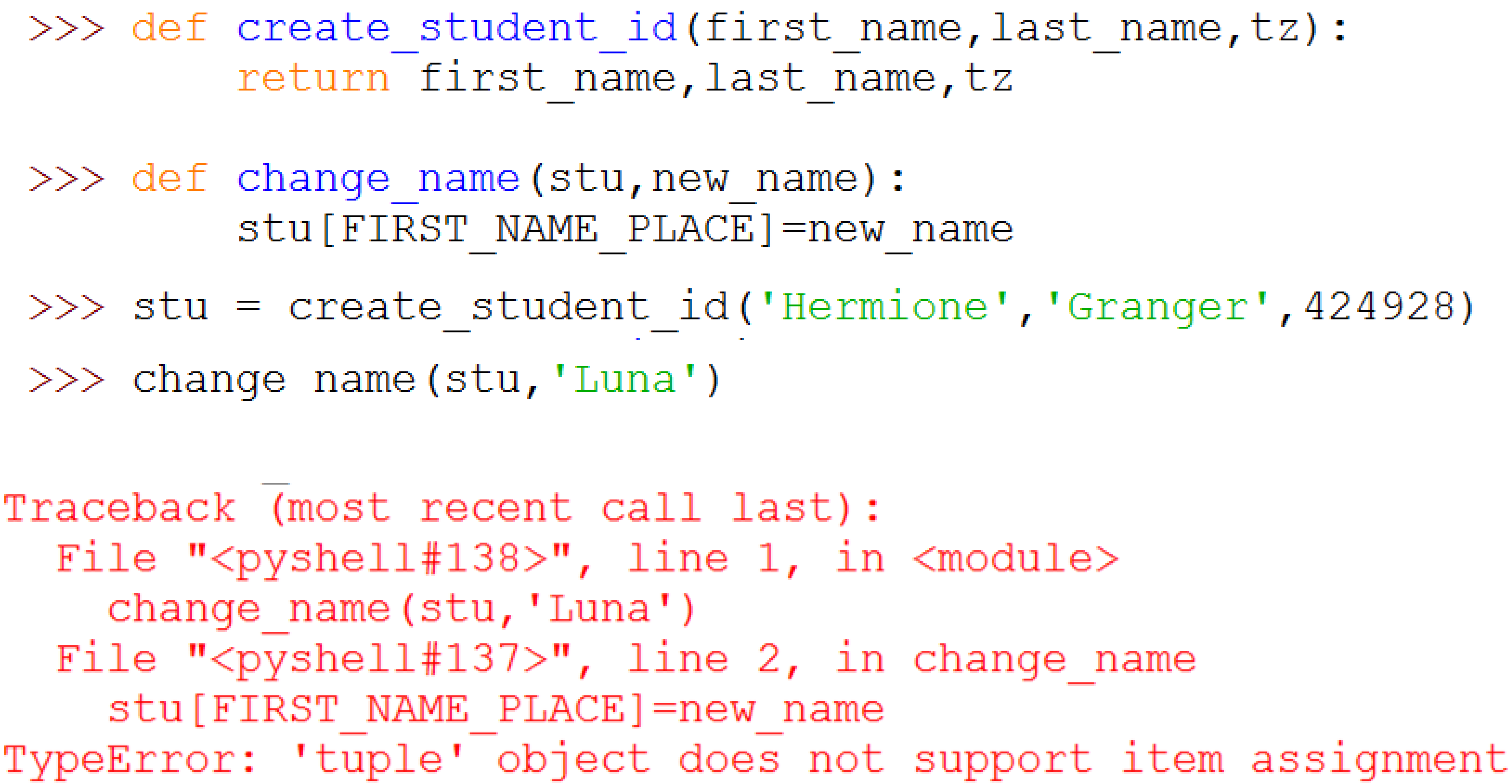
```
>>> grds=[90,97,87,65]
>>> modify_grades(grds,0,9)
>>> grds
[99, 97, 87, 65]
```



Huh?!?!
What happened to local variables?

Mutable items are passed as references!

Let’s test you



```
>>> def create_student_id(first_name,last_name,tz):
        return first_name,last_name,tz

>>> def change_name(stu,new_name):
        stu[FIRST_NAME_PLACE]=new_name

>>> stu = create_student_id('Hermione','Granger',424928)

>>> change name(stu,'Luna')

Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    change_name(stu,'Luna')
  File "<pyshell#137>", line 2, in change_name
    stu[FIRST_NAME_PLACE]=new_name
TypeError: 'tuple' object does not support item assignment
```

List Comprehension

List comprehension implementation

Mathematical notation:

$$S = \{\, x^2 \mid x = 1\ldots N \}$$

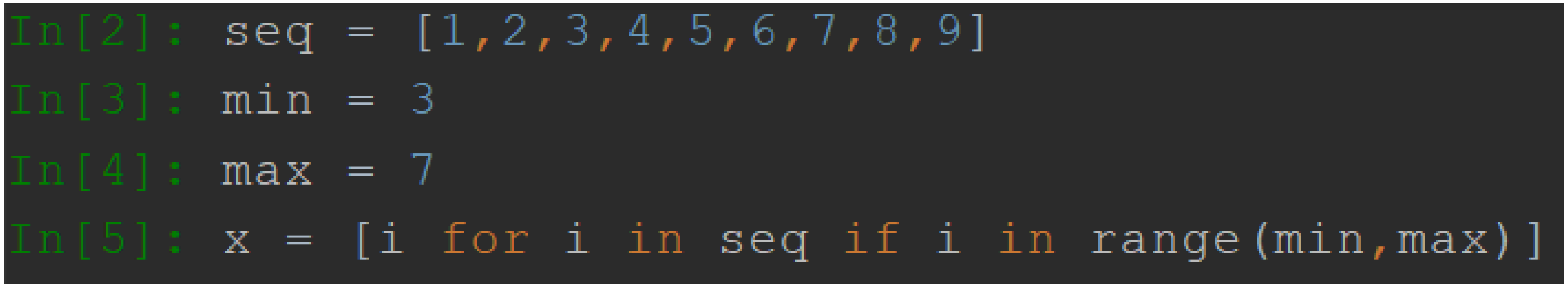
```
l = [exp(i) for i in seq if cond(i)]
```

```
l = []
for i in seq:
    if cond(i):
        l.append(exp(i))
```

Conditioning in List Comprehensions

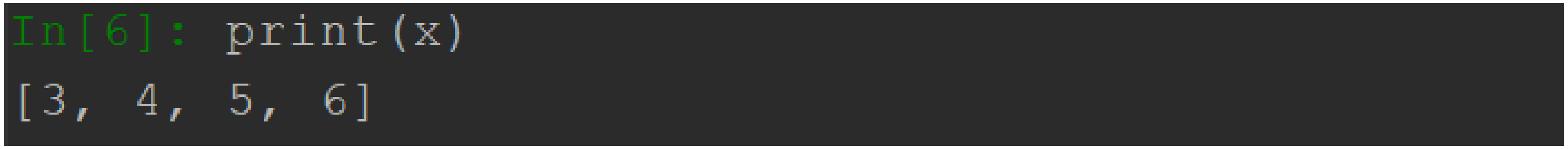
- You can add conditions inside an expression, allowing for the creation of a sub-list in a more “pythonic” way.

- Example:



```
In[2]: seq = [1,2,3,4,5,6,7,8,9]
In[3]: min = 3
In[4]: max = 7
In[5]: x = [i for i in seq if i in range(min,max)]
```

- The output is:

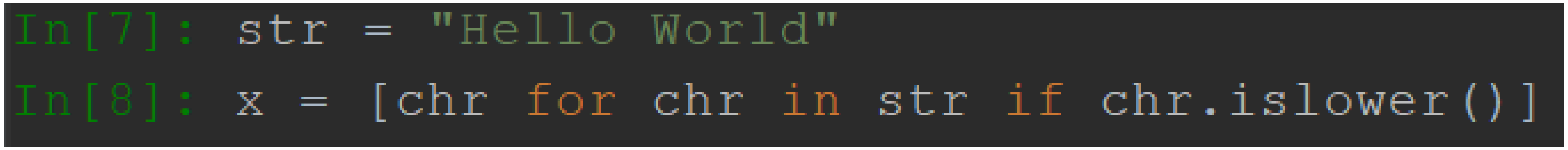


```
In[6]: print(x)
[3, 4, 5, 6]
```

- All numbers in seq within the range (min, max)

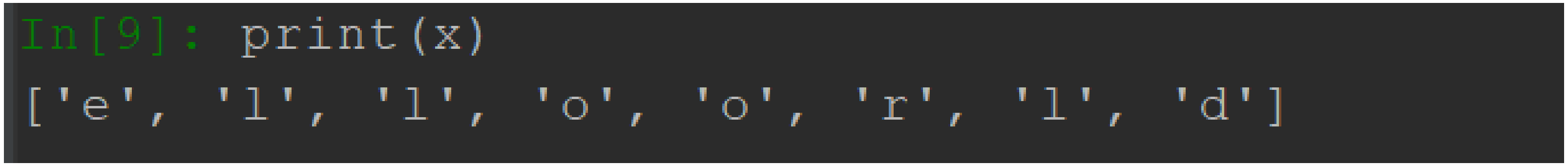
Conditioning in List Comprehensions

- Example:



```
In[7]: str = "Hello World"
In[8]: x = [chr for chr in str if chr.islower()]
```

- The output is:

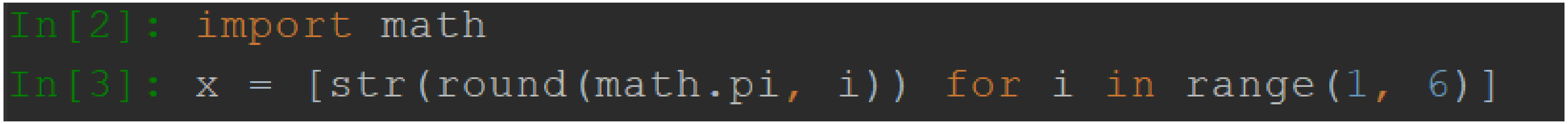


```
In[9]: print(x)
['e', 'l', 'l', 'o', 'o', 'r', 'l', 'd']
```

- All lower characters in a string

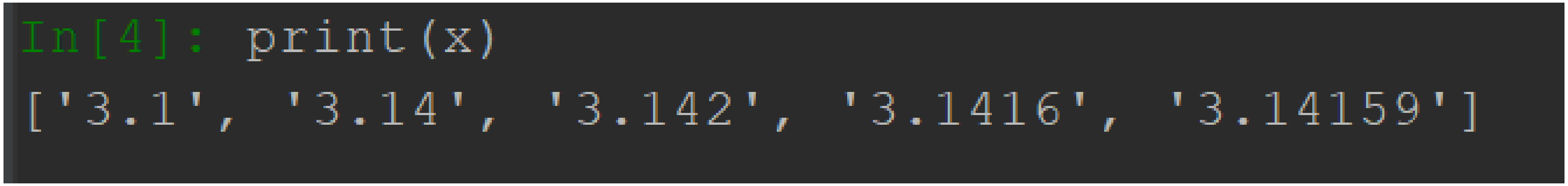
Conditioning in List Comprehensions

- Example:



```
In[2]: import math
In[3]: x = [str(round(math.pi, i)) for i in range(1, 6)]
```

- The output is:

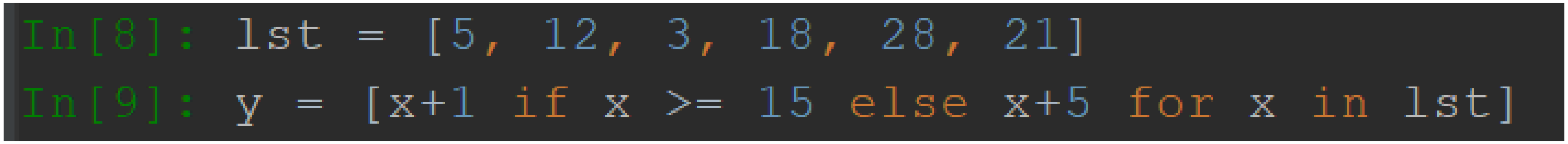


```
In[4]: print(x)
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

- Rounding of pi to $1 \leq i < 6$ numbers after the dot

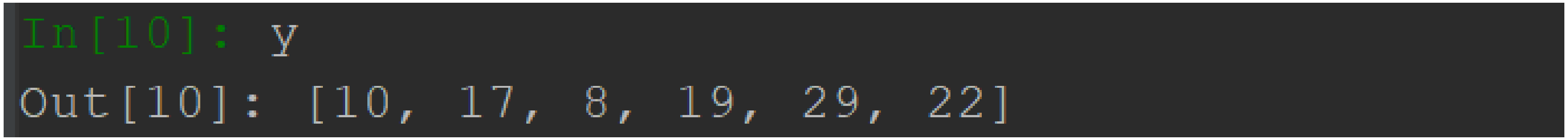
If - else Conditions in List Comprehensions

- Example:



```
In[8]: lst = [5, 12, 3, 18, 28, 21]
In[9]: y = [x+1 if x >= 15 else x+5 for x in lst]
```

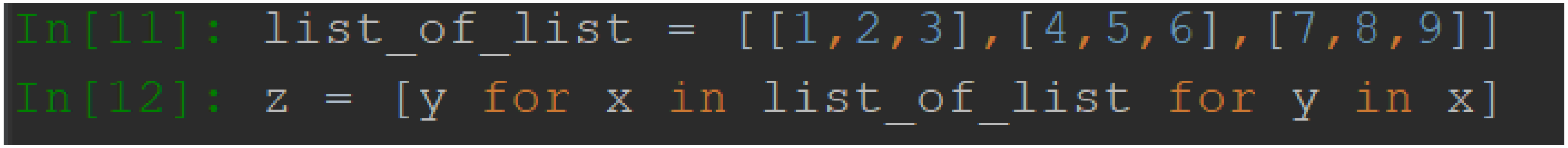
- The output is:



```
In[10]: y
Out[10]: [10, 17, 8, 19, 29, 22]
```

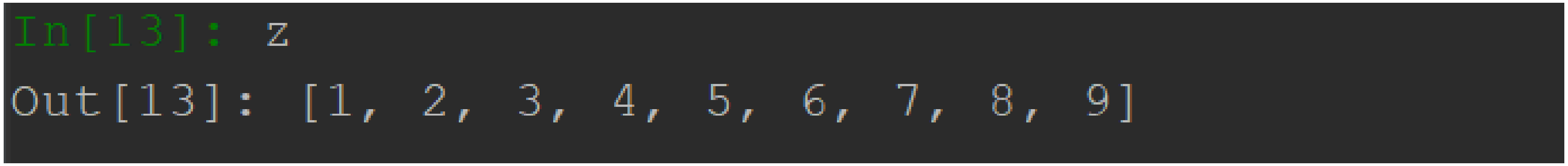
Nested List Comprehensions

- Example:



```
In[11]: list_of_list = [[1,2,3],[4,5,6],[7,8,9]]
In[12]: z = [y for x in list_of_list for y in x]
```

- The output is:



```
In[13]: z
Out[13]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Flatten ‘list_of_list’

Indexing and Slicing - Short recap

Indexing and Slicing - Strings

- Note how the start is always included, and the end always excluded

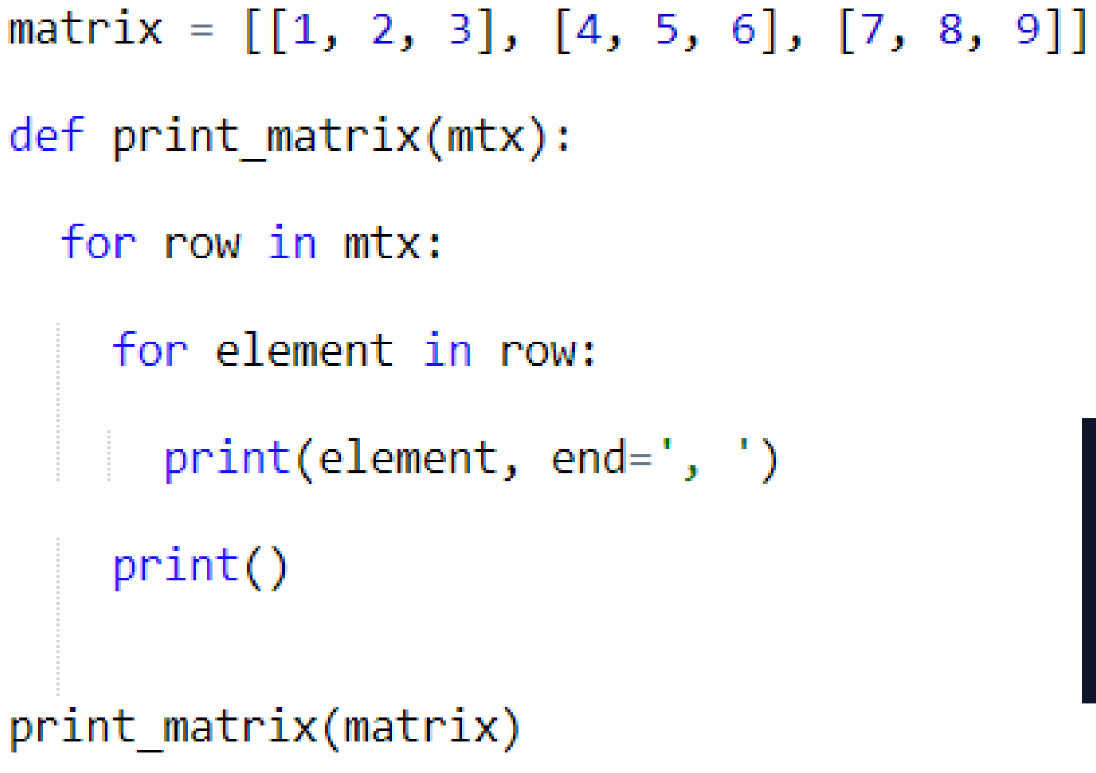
```
>>> word = ‘python’
>>> word[:4] + word[4:]      # same as word[::]
‘python’
```

- Slice indices have useful defaults:
  - an omitted first index defaults to zero,
  - an omitted second index defaults to the size of the string being sliced.

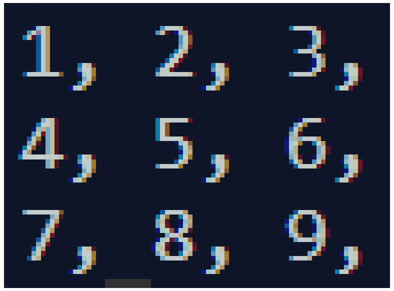
```
>>> word[:2]        # character from the beginning to position 2 (excluded)
 'Py’
>>> word[4:]        # characters from position 4 (included) to the end
'on’
>>> word[-2:]       # characters from the second-last (included) to the end
 'on‘
>>> word[::-1]      # characters from the end to the beginning
‘nohtyp’
```

Matrix example

- Suppose you are given a matrix:



```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

def print_matrix(mtx):

    for row in mtx:

        for element in row:

            print(element, end=', ')

        print()

print_matrix(matrix)
```

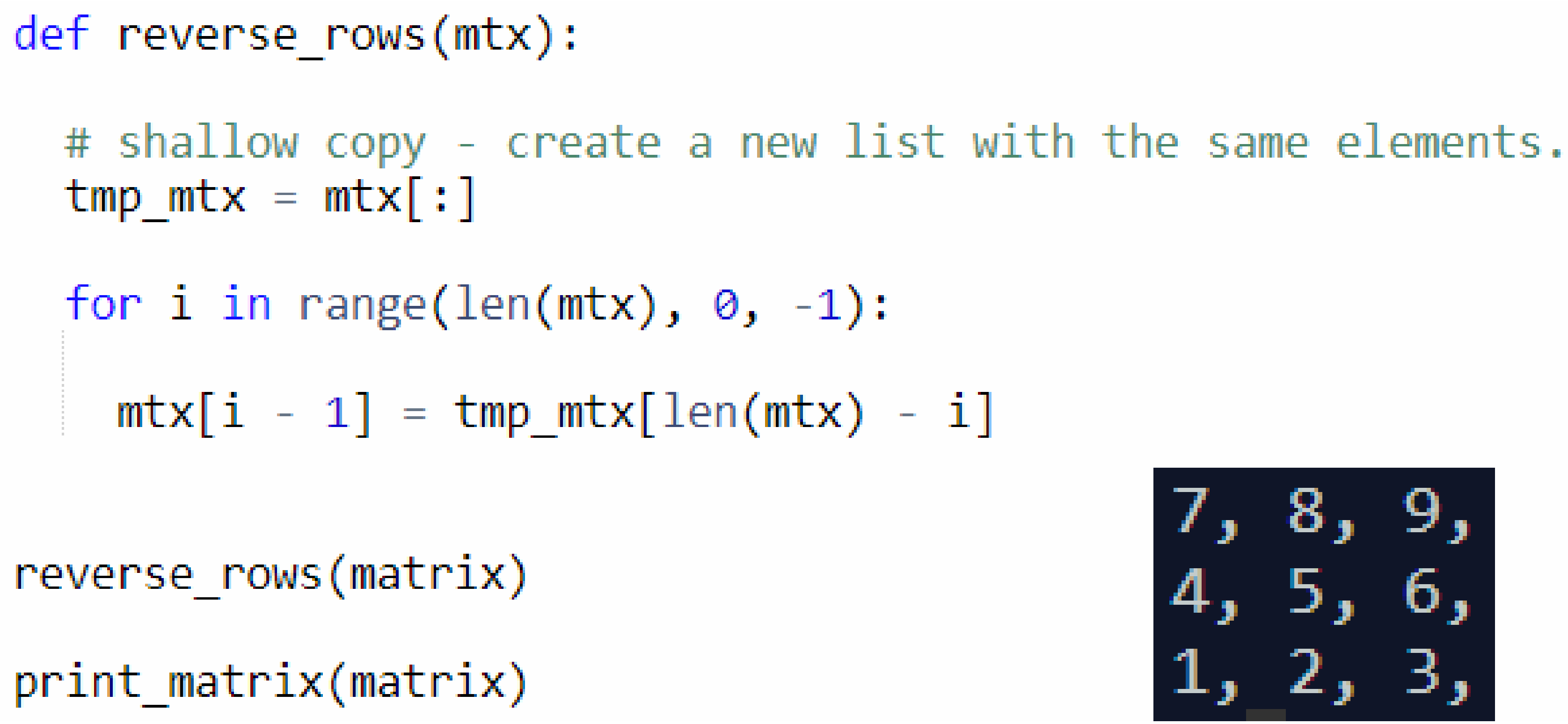


```
1, 2, 3,
4, 5, 6,
7, 8, 9,
```

- How would you reverse the rows?

Matrix reversing

- Reverse the rows’ references!

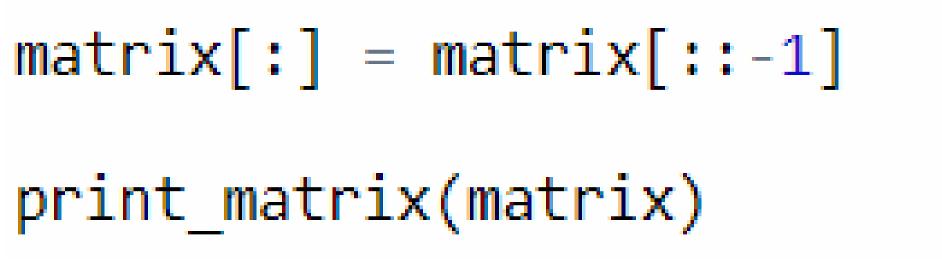


```
def reverse_rows(mtx):

  # shallow copy - create a new list with the same elements.
  tmp_mtx = mtx[:]

  for i in range(len(mtx), 0, -1):

    mtx[i - 1] = tmp_mtx[len(mtx) - i]


reverse_rows(matrix)

print_matrix(matrix)
```

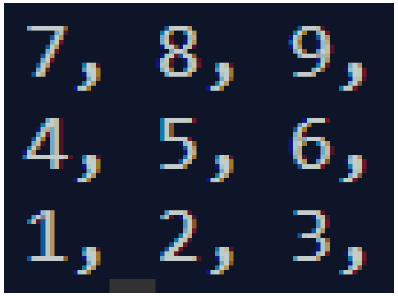
```
7, 8, 9,
4, 5, 6,
1, 2, 3,
```

Matrix reversing

- Actually, can be done simpler (3 operations):



```
matrix[:] = matrix[::-1]

print_matrix(matrix)
```



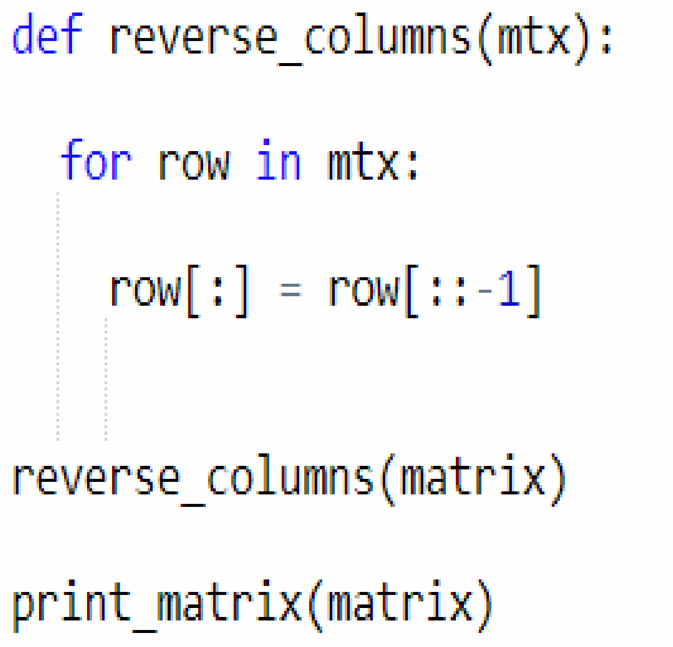
```
7, 8, 9,
4, 5, 6,
1, 2, 3,
```

- Replaced all **rows** with the reverse rows and kept the original matrix (list) object (same id!), just like the previous example.

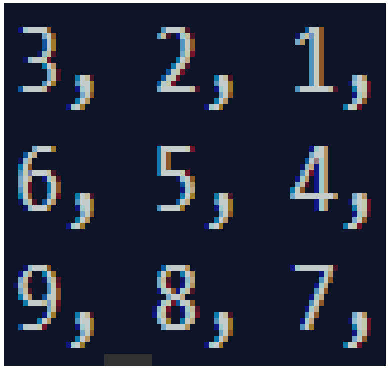
Matrix reversing

- How would you reverse the columns?

- Reverse each row (9 operations)!



```
def reverse_columns(mtx):
    for row in mtx:
        row[:] = row[::-1]

reverse_columns(matrix)
print_matrix(matrix)
```



```
3, 2, 1,
6, 5, 4,
9, 8, 7,
```