ISE 101 – Introduction to Information Systems

- Lecture 5 Objectives:
  - Functions
  - File I/O

FUNCTIONS

Functions

- Programs we have seen until now has a single body
- Most programs have to repeat the same procedure with different arguments
- functions are used:
  - Code reuse:

    The function is added to the software library.

    It is used later in other codes
  - Code maintenance

    Codes that are not well-written are very hard to maintain.

Functions

- You should not write the same piece of code many times scattered in the program (code dublication)
  - Hard to maintain
  - Larger and inpractical code
  - Hard to make changes
- The part of the program that implements a function is called a function definition
- When a function is subsequently used in a program, the function is called or invoked

Example

- Write a Python script that computes the average grade for student midterms
  - If midterm 1 grade is greater than 50 ,the average should be computed as

    average grade = 0.2*midterm 1+0.3*midterm 2+0.5*final
  - If midterm 1 grade is less than or equal to 50 and midterm 2 grade is greater than 50, the average grade should be computed as

    average grade = 0.3*midterm 1+0.3*midterm 2+0.4*final
  - Otherwise, the average grade should be computed as

    average grade = 0.4*midterm 1+0.3*midterm 2+0.3*final

Implementation 1

```
if midterm1_grade>50:
    average_grade=0.2*midterm1_grade \
        + 0.3*midterm2_grade + 0.5*final_grade
elif midterm1_grade<=50 and midterm2_grade>50:
    average_grade=0.3*midterm1_grade \
        + 0.3*midterm2_grade + 0.4*final_grade
else:
    average_grade=0.4*midterm1_grade \
        + 0.3*midterm2_grade + 0.2*final_grade

print(average_grade)
```

Implementation 1

- Grade averaging is used many times at different exercises
- Instead of writing the average equation each time, a function should be used.
- The arguments of the function should be
  - Exam grades
  - Grade weights

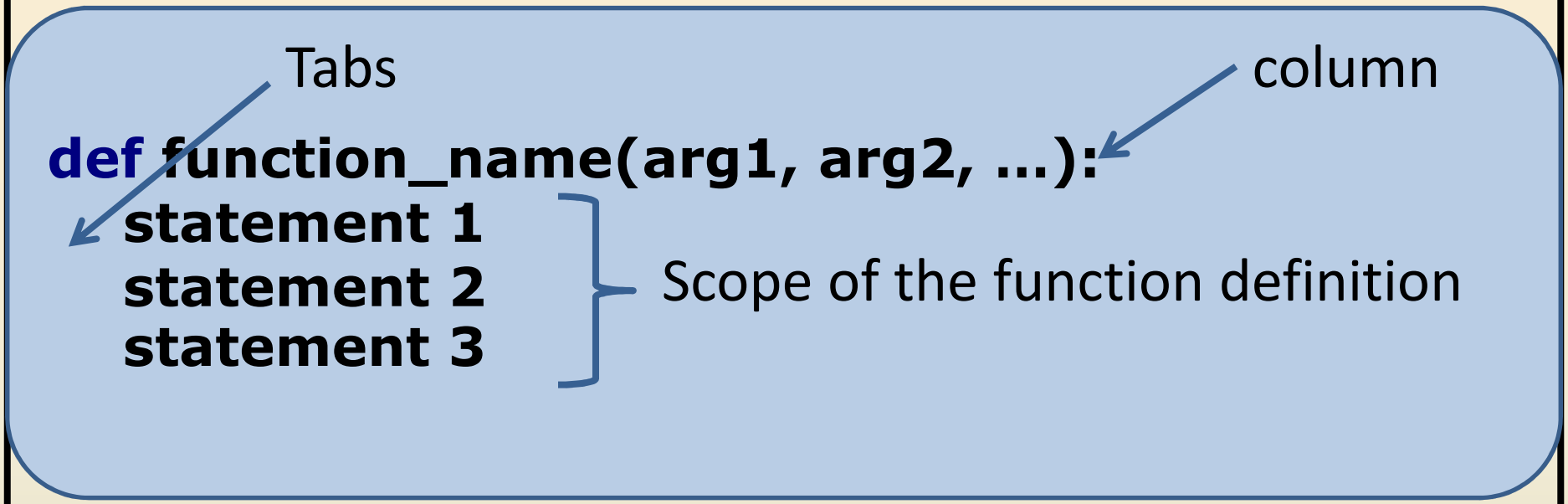
Implementation 2

```
def average_grade(midterm1_grade, weight1,
          midterm2_grade, weight2, final_grade, weight3):
    average_grade=weight1*midterm1_grade \
        + weight2*midterm2_grade + weight3*final_grade
    print(average_grade)
```

```
if midterm1_grade>50:
    average_grade(midterm1_grade,0.2,midterm2_grade,0.3,
                  final_grade,0.5)
elif midterm1_grade<=50 and midterm2_grade>50:
    average_grade(midterm1_grade,0.3,midterm2_grade,0.3,
                  final_grade,0.4)
else:
    average_grade(midterm1_grade,0.4,midterm2_grade,0.3,
                  final_grade,0.3)
```

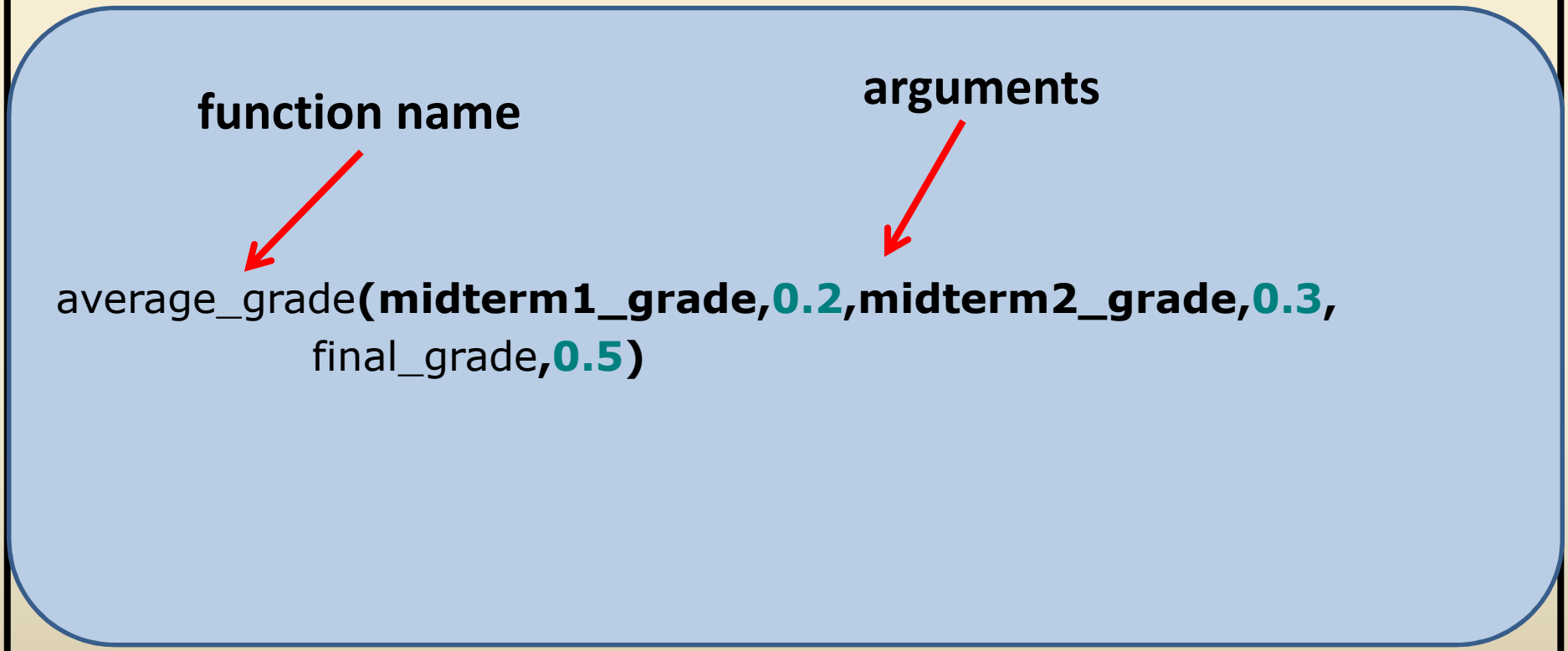
Function Definition

- Structure of function definition



Functions

- Functions can be called by simply writing their name and arguments inside parenthesis



Functions

- The idea of the functions is to repeat the same procedure with different parameters
- Therefore, functions can take parameters (or arguments)
- These parameters are defined in the function definition

```
def myFunction(name,age):
  print("Welcome ",name)
  print("Next year you will be: " + str(age+1)
                                   + "years old.")
```

- In this example,

  name is the first argument

  age is the second argument

Functions

- When Python calls a function
  - The calling program suspends execution at the point of the call
  - Function arguments are passed to the function
  - The scope of the function is executed
  - Control returns to the point just after where the function was called

Functions

- To call the function, the arguments have to be given in the correct order

```
>>> myFunction('Ali',13)
Welcome  Ali
Next year you will be: 14years old.
>>> myFunction(13,'Ali')
Welcome  13
Traceback (most recent call last):
builtins.TypeError: Can't convert 'int'
object to str implicitly
```

Functions

- Function has to be called with the exact number of arguments that are used in the definition

```
>>> myFunction('Ali',5)
Welcome  Ali
Next year you will be: 6 years old.
>>> myFunction('Ali',5,3)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
builtins.TypeError: myFunction() takes
exactly 2 positional arguments (3 given)
```

Argument Passing

- Pass by value:
  - local copy of the variable is generated and sent to the function
  - If the local copy is changed within the function, the variable is not changed outside the scope of the function

Pass by Value

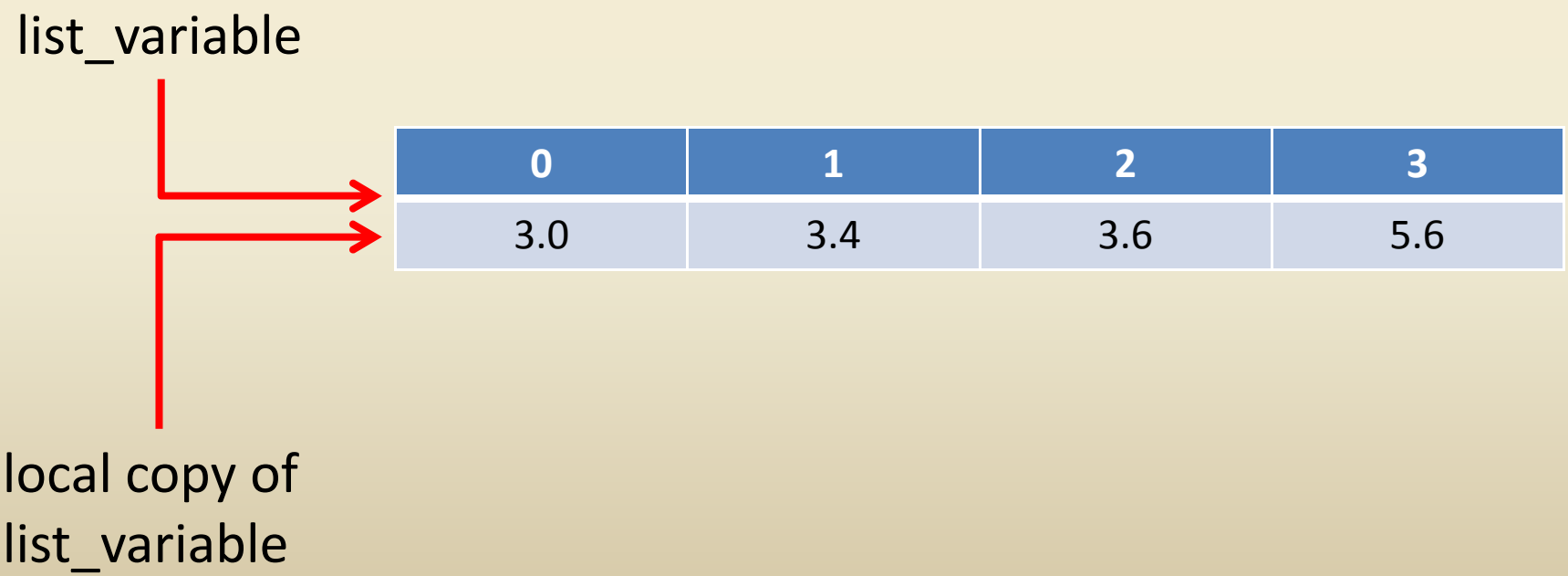
```
def increment(x):
    print('incoming x: ' + str(x))
    x=x+1
    print('changed x: ' + str(x))

x=5
print('before function call x: ' + str(x))
increment(x)
print('after function call x: ' + str(x))
```

```
before function call x: 5
incoming x: 5
changed x: 6
after function call x: 5
```

Argument Passing

- Pass by reference:
  - local copy of the pointer is generated and sent to the function
  - Location that it points can be changed



Pass by Reference

```
def increment(x):
    print('incoming x: ' + str(x))
    for i in range(len(x)):
        x[i]=x[i]+1
    print('changed x: ' + str(x))

x=[1, 2, 4]
print('before function call x: ' + str(x))
increment(x)
print('after function call x: ' + str(x))
```

```
before function call x: [1, 2, 4]
incoming x: [1, 2, 4]
changed x: [2, 3, 5]
after function call x: [2, 3, 5]
```

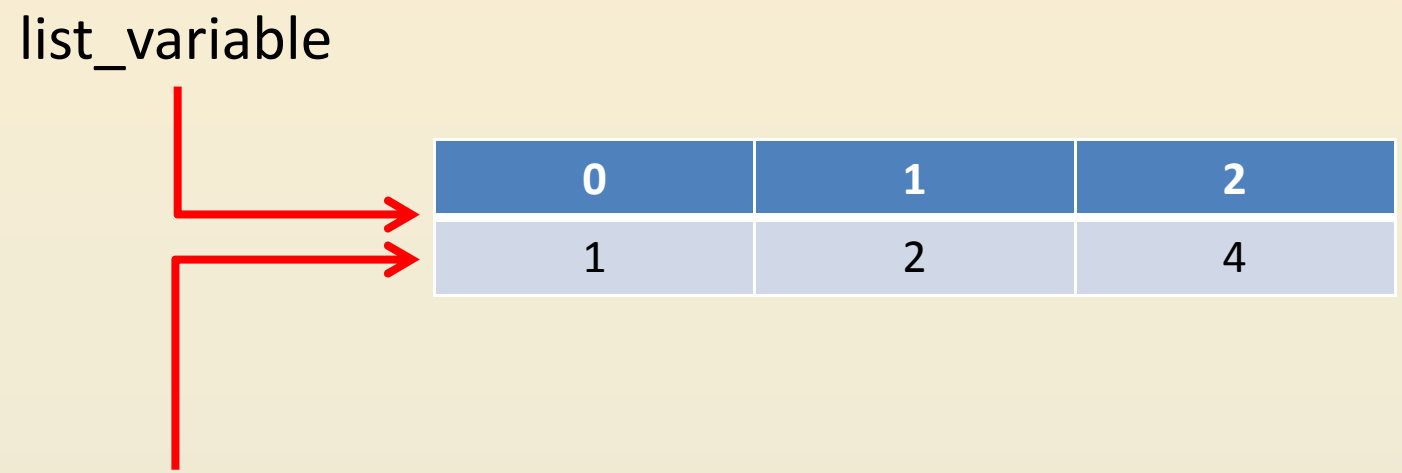
Pass by Value

```
def increment(x):
    print('incoming x: ' + str(x))
    x=[2,3,5]
    for i in range(len(x)):
        x[i]=x[i]+1
    print('changed x: ' + str(x))

x=[1, 2, 4]
print('before function call x: ' + str(x))
increment(x)
print('after function call x: ' + str(x))
```

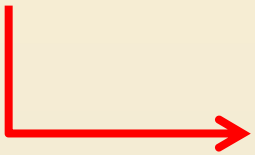
```
before function call x: [1, 2, 4]
incoming x: [1, 2, 4]
changed x: [3, 4, 6]
after function call x: [1, 2, 4]
```

- In the beginning of the function

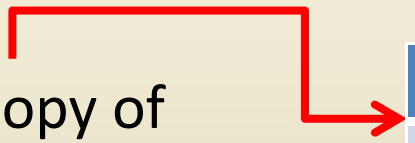


- After `x=[2,3,5]` in the function

list_variable



0	1	2
1	2	4

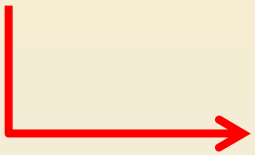


local copy of list_variable

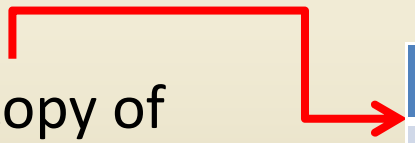
0	1	2
2	3	5

- After the for loop in the function

list_variable



0	1	2
1	2	4



local copy of list_variable

0	1	2
3	4	6

Return Values

- Sometimes the functions produce results
- These results can be returned to the code that calls the function
- “return” expression is used to return value
- Contrary to other programming languages Python can return more than one value

Return Value

```
def power(val,exp):
    result=1
    for i in range(exp):
        result=result*val
    return result

x=power(2,8)
print(x)
```

Returning Multiple Values

- Functions can return multiple values that are separated by commas
- return val1, val2, val3, …

```
def add_subtract(x1,x2):
    sum=x1+x2
    dif=x1-x2
    return sum,dif

x,y=add_subtract(3,4)
print(x)
print(y)
```

Returning Multiple Values

- If a function returns multiple values, the returning values should be assigned to the exactly same number of parameters

```
def add_subtract(x1,x2):
    sum=x1+x2
    dif=x1-x2
    return sum,dif

x=add_subtract(3,4)
x,y,z=x=add_subtract(3,4)
```

x will be a tuple

Error

“global” Statement

- If you do not want to use the local value of a variable, “global” statement is used
- When “global” is used for a variable within a function, its value that is assigned outside the scope is used

```
x = 50
def func():
  global x
  print('x is', x)
  x = 2
  print('Changed global x to', x)

func()
print('Value of x is', x)
```

“global” Statement

```
x = 50
def func():
  global x
  print('x is', x)
  x = 2
  print('Changed global x to', x)

func()
print('Value of x is', x)
```

```
x is 50
Changed global x to 2
Value of x is 2
```

“global” Statement

- Do NOT use “global”
- It is a not a good programming practice
- Functions with “global” statement cannot be reused
- Functions are used for code reusability
- “global” contradicts with philosophy of functions and code reusability

“nonlocal” Statement

- “nonlocal” statement uses the variable in the outer scope
- “global” and “nonlocal” are not the same
  - “global” has a single and global scope
  - “nonlocal” uses the variable that is outside the local scope

Example

```
def func_outer():
  x = 2
  print('x is', x)
  def func_inner():
    x = 5
  func_inner()
  print('Changed local x to' + str(x))

x=7
func_outer()
print('Value of x: ' + str(x))
```

```
x is 2
Changed local x to 2
Value of x: 7
```

Example

```
def func_outer():
  x = 2
  print('x is', x)
  def func_inner():
    global x
    x = 5
  func_inner()
  print('Changed local x to ' + str(x))

x=7
func_outer()
print('Value of x: ' + str(x))
```

```
x is 2
Changed local x to 2
Value of x: 5
```

Example

```
def func_outer():
  x = 2
  print('x is', x)
  def func_inner():
    nonlocal x
    x = 5
  func_inner()
  print('Changed local x to ' + str(x))

x=7
func_outer()
print('Value of x: ' + str(x))
```

```
x is 2
Changed local x to5
Value of x: 7
```

Default Argument Values

- Some of the arguments can be assigned default values
- If a default value is assigned to a variable, that argument becomes optional
  - If the argument is not sent to the function, the default value of the argument is used
  - If the argument is sent, this value is used
- The default value should be immutable
- Default arguments should be placed after other (non default) arguments

Example

```
def func(a, b=3, c=10):
    print('a is ' + str(a))
    print('b is ' + str(b))
    print('c is ' + str(c))

func(1, 5, 7)
```

```
a is 1
b is 5
c is 7
```

Example

```
def func(a, b=3, c=10):
    print('a is ' + str(a))
    print('b is ' + str(b))
    print('c is ' + str(c))

func(6)
```

```
a is 6
b is 3
c is 10
```

Example

```
def func(a, b=3, c=10):
    print('a is ' + str(a))
    print('b is ' + str(b))
    print('c is ' + str(c))

func(6,7)
```

```
a is 6
b is 7
c is 10
```

Example

- Write a Python function (not a script) named "inner_product" that takes 2 lists as arguments and returns their inner product.

```
def inner_product(x1,x2):
    sum = 0
    if len(x1)!=len(x2):
        print("Warning: lists have different lengths")
        return 0

    for i in range(len(x1)):
        sum=sum+x1[i]*x2[i]

    return sum
```

Example

```
def inner_product(x1,x2):
    sum = 0
    if len(x1)!=len(x2):
        print("Warning: lists have different lengths")
        return 0

    for i in range(len(x1)):
        sum=sum+x1[i]*x2[i]

    return sum

t1=[1,1,1]
t2=[2,3,6]
print( inner_product(t1,t2) )
```

Example

- Write a Python function named “find_max” that takes a list as argument and returns the maximum value within this list.

```
def find_max(x):
    max_value=x[0];
    for i in range(1,len(x)):
        if x[i]>max_value:
            max_value=x[i]

    return max_value


t2=[2,3,6]
print( find_max(t2) )
```

Example

- Write a Python function named “convert_seconds” that takes the number of seconds as argument and returns the equivalent hour/minute/second.

```
def convert_seconds(nseconds):
    nhours=int(nseconds/3600)
    tmp=nseconds%3600
    nminutes=int(tmp/60)
    nseconds=residue%60

    return nhours,nminutes,nseconds

nseconds=123213
h,m,s=convert_seconds(nseconds)

print(str(nseconds) + ' seconds = ' + str(h)
                          +':'+str(m)+':'+str(s))
```

FILE I/O

File IO

- Frequently data should be read from a file on the hard drive
- Results should be written to a file in the hard drive
- File input/output is important

File IO

- A file is a sequence of data that is stored in disk
- Files can contain any data type
- Text-files contain text. They can be thought as a long string (of many lines)
- Files have a special characters to denote the end of lines and end of file
- These special characters help us to parse the files
- Different programming languages have nearly the same concept of file processing
  - Files are opened using a mode (read, write, append)
  - Data in the file is processed
  - Files are closed

File IO

- File open

  <filevar>=open(<filename>,<mode>)
- filevar is a handle that will be used for further file operations such as reading/writing etc.
- filename is the name of the file on OS
- mode is a string
  - “r” for reading
  - “w” for writing
- If no mode is given, default mode is “r”
- When the file processing is finished, close it

  <filevar>.close()
- What happens if a file is not closed?

File IO

- Some simple functions for reading files

<filevar>.read()	returns the entire remaining contents of the file as a single string
<filevar>.readline()	returns the next line of the file. That is all text up to and including the newline character
<filevar>.readlines()	returns a list of the remaining lines in the file. Each list item is a single line including the newline character at the end

File IO

- <filevar>.read() may lead to very long strings which will be stored on the memory of the computer.
- This may slow down the computer
- You should prefer reading a file line-by-line and process each line seperately
- What does this code do?

```
>>> infile=open("list.txt","r")
>>> for i in range(5):
        line=infile.readline()
        print line[:-1]
>>> infile.close()
```

File IO

- Python treats the file as a sequence of lines. Looping through the lines of a file can be done directly as:

```
>>> infile=open("list.txt","r")
>>> for lines in infile:
        print lines[:-1]
>>> infile.close()
```

File IO

- Opening a file for writing prepares that file for receiving data.
- If file does not exist, it is created
- If the file exists, it is DELETED
- Opening a file for writing

  <filevar>=open(<filename>,”w”)
- Data can be written into the file as

  <filevar>.write(<string>)
- write function is similar to print. But it is not as flexible
  - takes a single string argument
  - new line should be explicitly provided

File IO

- What does this code do?

```
>>> ofp=open('list.txt','w')
>>> ofp.write('First line\n')
>>> for i in range(10):
        ofp.write('this is line %d\n' % i)
>>> ofp.close()
```

- Output

```
First line
this is line 0
this is line 1
this is line 2
this is line 3
this is line 4
this is line 5
this is line 6
this is line 7
this is line 8
this is line 9
```

Example

```
# this is a comment line discard this line
# records start from here
04001020 ; Ali Gel; 20 ; 34; 100
04001032 ; Veli Git; 36 ; 23; 57
04002123 ; Ferhat Can; 44 ; 46 ;90
```

- Any line starting with a “#” is a comment line, your program should not process these lines
- Each record has 5 fields: student number, student name and surname,
- midterm 1 grade, midterm 2 grade, final exam grade.
- Fields are separated with semicolons.

Example

- Total grade is computed as
  - %25 from midterm 1
  - %35 from midterm 2
  - %40 from final exam
- Read these records from the given file
- Compute and display the
  - average grade of midterm 1
  - average grade of midterm 2
  - average grade of final exam

```
ifp=open('file.txt','r')

def find_average(grades):
    return(sum(grades)/len(grades))

midterm1_grades=[]
midterm2_grades=[]
final_grades=[]
for line in ifp:
    if line[0]=='#':
        continue
    info=line.split(";");
    midterm1_grades.append(int(info[2]))
    midterm2_grades.append(int(info[3]))
    final_grades.append(int(info[4]))

print('Avg for midterm 1:'+str(find_average(midterm1_grades) ))
print('Avg for midterm 2:'+ str( find_average(midterm2_grades) ))
print('Avg for final: ' + str( find_average(final_grades) ))

ifp.close()
```

Example

- Write a Python script that reads a file, replaces each newline with double newlines and writes the output to another file

```
ifp=open('file.txt','r')
ofp=open('output.txt','w')

for line in ifp:
    ofp.write(line)
    ofp.write('\n')

ifp.close()
ofp.close()
```