Python Notes **Elena**

Issue 3 12 January 2023

1) Data Types

Integer	int()	x = 1
Float	float()	x = 1.0
Complex	complex()	x = 1+1**j**
Range *sequence of integers*	range(start=0, stop, step=1), *stop is exclusive*	x =**range**(5) **for** i **in** x: :
Boolean	bool()	x = **True;** x= **False**
None	None	x = **None**
String	str()	x = 'abcd'
List	list	x = [1,2,3,4]
Tuple	tuple()	x = (1,2,3,4)
Dict	dict()	x = {'k1':1 , 'k2':2}
Set	set()	x = (1,2,3,4)

See Annex 1 for the properties of data types

2) Keywords and general functions

https://www.w3schools.com/python/python_ref_keywords.asp
in = boolean operator to check if an element is in a sequence / used in the **for** loop
pass = null statement
break = exits from a **for** or a **while** loop
any(*it*) = True if any item in the iterable *it* is true, otherwise it returns False
all(*it*) = True if all items in the iterable *it* are true or if the iterable is empty

Naming Conventions

Variable and Functions: *name_of_var* OR *name_of_function*
Class: *NameOfClass*

3) Functions/Procedures

Declaration of Function, use **def**

```
def fn1(p1): # p1 is a function parameter
    a = p1    # variable a and p1 are local
    print(a)
:
```

Call of a function

```
fn1(10)   # 10 is the argument of function fn1
>>>> 10
```

Scope of objects (Namespace): Global (main program or module), enclosing (function), local (sub-function)
https://realpython.com/python-namespaces-scope/#:~:text=the%20next%20level.-,Namespaces%20in%20Python,values%20are%20the%20objects%20themselves
globals() = returns a dictionary that contains all the global objects
locals() = returns a copy of the dictionary that contains all the local objects at the time of the call

Function Parameters and function variables are local

```
def fn1(p1): # p1 is a function parameter
        a = 10+p1     # variable a and p1 are local
        print(locals())
fn1(10)
>>>> {'p1': 10, 'a': 20}
```

When calling a function, arguments of the function call can be passed as positional or by keyword
(first positional in their order and then keywords in any order)

```
    def fn1(p1, p2, p3): # p1, p2 and p3 are parameters of fn1
        :
    fn1(1, 2, 3) # a,b,c are positional arguments
    fn1(1, p3=3, p2=2) # first positional argument  p1 and then keyword args p2 and p3 in any order
    fn1(p3=3, p2=2,p1=1) # only keyword args p1, p2 and p3 in any order
```

N.B. Elements of a List or dictionary passed as a parameter can be modified

```
    def fn1(x):
        x[0]='10'
        :
    l1=[1,2,3]
    fn1(l1) # l1 = [10,2,3]
```

Elements o List or dictionaries defined in the global or enclosing namespace can be modified by sub-functions

```
    a=[1,2,3] # list a is visible to all sub-functions
    def fn1():
        a[0]=100 # variable a of enclosing function or main is visible within fn1
        :
    fn1() # a=[100,2,3]
```

global var declares a variable global in a sub-function (difficult to debug)

```
    a=10
    fn1():
        global a
        a=1000
        :
    fn1() # a=1000
```

Variable length input parameters in the definition of a function: use of * for tuple and ** for dictionaries (see later packing and unpacking),
* gathers all input positional parameters into tuple *args*
** gathers keyword arguments into dictionary *kwargs* e.g.

```
    def fn1 (*args,**kwargs):
        a=args[0]
        b=kwargs[k1]
```

Pass multiple positional parameters when calling a function :
Use ***t** to unpack tuple **t**

```
    def fn2(a,b):
    :
    t=1,2
    fn2(*t)
```

Pass multiple keyword parameters when calling a function :
Use ****d** to unpack dictionary **d**

```
def fn2(a,b):
:
d={'a':1,'b':2) # or d=dict(a=1,b=2)
fn2(**d)
```

Optional Input parameter: assign default value

```
def fn3(a,c=20):
:
fn3(39)  # pass only a = 39 and set c=20
```

Return parameters and expressions
return tuple with values of variables

```
return a,b,c
```

return the value of an expression

```
return a+b+c
```

4) Boolean

Values	**True, False**	
Logical Operations	**and, or, not**	
Comparisons operations	**<** **<=** **>** **>=** **==** **!=** **is** **is not**	Different object identity negated obj. identity

5) String

s='aaaa'
String Iterable, non-Unique, immutable, Hashable and Ordered

String Operations
s **+** *s1* = concatenate strings *s* and *s1*
s ***** n = repeat string *s* n times

% String format operator (old style, C legacy)
fs % *values* = % format tuple *values* using string *fs,*
fs contains one one more % with conversion parameters
number of elements in *values* must be equal to number of % in fs
Example:

```
# format string, with 2 input arguments
# % = argument,
# d and s are conversion types, decimal and string
fs='decimal: %d, string: %s'
# tuple with 2 input arguments
t=(35,'Test')
print(fs%t)
>>>>>> decimal: 35, string: Test
```

Conversion Type	Meaning
'd' or 'i'	Signed integer decimal.
'o'	Signed octal value.
'x'	Signed hexadecimal (lowercase).
'e'	Floating point exponential format (lowercase).

'f'	Floating point decimal format.
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
's'	String (converts any Python object using str()).

more details on string format operator in Annex 2

f String format operator (new style)
https://realpython.com/python-formatted-output/

```
var1=10
var2="aaaaaa"
print(f"text1={var1}, text1={var2}")
>>> text1=10, text2=aaaaaa
```

String Methods
list_of_strings = s.**split**(*delimiter*), default is white-space.
s1 = delimiter.**join**(*iterable_string*)
s.**find**(*s1*,*i1*,*i2*)= find string *s1* in s searching from *i1* to *i2* (excluded)
s.**strip**() removes newline \n
—

s.**count**(*value, start, end)*
returns the number of times *value* appears in the string *s*
s.**replace**(*oldvalue, newvalue, count*)
replaces *oldvalue* with *newvalue* for *count* times

String Functions
len(string) length of the string
str(*object*) converts an object into a string

Escape characters (for special actions when writing on a device the string)

Code	Result	Example
\'	Single Quote	'It\'s' >> It's
****	Backslash	' \\' >> \
\n	New Line	' line1 \n line2' line1 line2
\r	Carriage Return	'a\rb' >> b
\t	Tab	'a\tb'
\b	Backspace	'abc\bD' >> abD
\f	Form Feed	'abc\fD' >> abc D
\ooo	Octal value	'\123' >> 5
\xhh	Hex value	'\x48' >> H

6) Lists
l1 = [*el1,el2,..*]
Lists are Iterable, non-Unique, Mutable, not-Hashable and Ordered

List Operations
l1=[] empty list initialization
l1**+**l2 = concatenate lists l1and l2
l1*****n = repeat list n times
del l1(i1:i2) = delete elements of list l1 from index i1 to i2 (excluded)
l2=l1[i1:i2] l2 is a new list

copy one by one of the elements of l1 from index i1 to i2 (excluded) into l2,
l2=l1[:] l2 is a new list
Complete copy of all elements of l1 (also multidimensional) to l2,
l2[:] = l1 copy all the elements of l1 into l2 one by one,
N.B. list l2 must be already existing
l2=l1 copy id (i.e. address) of l1 on l2.
N.B. l2=l1, l1[2]='a' is equivalent to l2[2]='a'
they both change the same element in memory
m **in** l = test if m is a member of list l
If l1: # test if l1 is not empty (l1 !=[])

List comprehension:
generates a new list processing an iterable according to a condition
newlist = [*expression* for *item* in *iterable* **if** *condition* **==** True]

List Functions
sum(l1)
len(l1)
sorted(s)= sorted list of the sequence
list(iterable_object) transform iterable object into a list

List Methods
l1.**append**(*elem*) = append element elem to list *l1*
l1.**clear**()
l1=l.**copy**() (for nested data structures)
l1.**count**(value)
l1.**extend**(*l2*) append all the elements of list *l2* to list *l1*
index=l1.**index**(value) # find index of 1st element of l1 equal to value
l1.**insert**(*index*, value)
insert a new element with content value in list l1 at position index
element=l1.**pop**(index) remove element of list l1 at position index
l1.**remove**(value) = remove first element of list l1 with value
l1.**reverse**()
l1.**sort**(reverse=**True**|**False**, key=myFunc)
def my_funct(el): # example
sort by 2 operators on the element el, fn1 and fn2 in order
return (fn1(el),fn2(el))
e.g. **return**(len(el),el) # by length of element and then alphabetic or numerical order

7) Dictionary

d={key1:value1, key2:value2, ...}
Dictionaries are Iterable, Unique(keys), mutable, not-Hashable and ordered
d[key1] returns value1
Iterator returns key, e.g. **for** el **in** d: provides key1, key2 ..
Order is unpredictable
Key is immutable
Contains key-value pair {key:value}
Keys can be tuples

Dictionaries Functions
dict(*iter*) = creates a dictionary using the iterable *iter,*
iter must return a 2 elements tuple with key and value
e.g. key and values contained in the tuple of tuples *t_t*

```
# a and b are the keys, 1 and 2 are the values
t_t=(('a',1),('b',2))
dict(t_t)
```

e.g. single key and value contained in the tuple of tuples *t_t*

```
t_t=(('a',1),)  # N.B. one element tuple needs the comma
dict(t_t)# a is the key, 1 is the value
```

e.g. specification of keys and values as input

```
dict(a=1,b=2)
```

len(d)

1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

Dictionaries Methods

d.**items**() = returns iterator of tuple (key:value)

d.**get**(*keyname, defaultvalue*) = Returns the value of the specified key or optional value if key does not exist

d.**clear**() Removes all the elements from the dictionary

d1 = d.**copy**() Returns a copy of the dictionary

d1 = d.**fromkeys**((*keys, value*)

Returns a dictionary with the specified keys and value

d.**keys**() Returns a list containing the dictionary's keys

value = d.**pop**(*key*,*defaultvalue*) Removes the element with the specified key

d.**popitem**() Removes the last inserted key-value pair

value = *d*.**setdefault**(*keyname, value*)= Returns the value of the specified key.

If the key does not exist: insert the key, with the specified value

d.**update**(*iterable*) Updates the dictionary with *iterable* with key-value pairs,

e.g.

```
l_t=[('a',100)] # list of tuples
d.update(l_t)     # insert or update key 'a' with value 100
```

d.**values**() = Returns a list of all the values in the dictionary

8) Tuple

t=(el*1,el2,..*)

Tuple is Iterable, Immutable, not-Unique, Hashable and ordered

Created by using commas and ()

```
t=(1,2,3)
t = 1, 2, 3
t=(1,2,3,)
```

N.B. 1 element tuple MUST end with comma

```
t=1,
t=(1,)
```

tuple(*iter*) Convert iterable *iter* a sequence into a tuple

Tuples can be joined by adding, +

```
(1,2,3) + (10,20,30)
>>>> (1,2,3,10,20,30)
```

Useful for swap, multiple assignments, return of multiple values, image storing

9) Packing & unpacking of tuples (*) & dictionaries (**)

*el = operator on element el, transforms an iterable into a list

Gathering/Packing of element

used in assignment (left side) gathers elements of an input iterable into a new list, e.g.

```
(x,*y,z)=(1,2,3,4)  # x=1, y=[2,3] and z=4
```

Scattering/Unpacking of the tuple

used in assignment (right side) scatters a tuple in a new list , e.g

```
t=(1,2,3)
a=(*t,) # a=(1,2,3)
b=[*t] # b=[1,2,3]
```

**d = operator on dictionary d

Gathering of the dictionary
used in assignment (right side)
scatters the elements of the input dictionary into the new dictionary,
e.g for merging dictionaries

```
d1={'a':20,'b':100}
d2={'c': 3, 'd': 300}
d3={**d1,**d2}
# d3 is a new dictionary with the content of d1 and d2 ,
# d3 = {'a':20, 'b':100, c': 3, 'd': 300}
```

10) Set

A set is Iterable, not-mutable (only add or remove elements), Unique, not-Hashable and unordered
Set is created using curly brackets
s={*e1*,*el2*,}
s=**set**(*iterable*), e.g. to create {e1,el2,..}

```
set([el1,el2,  ..])
set((el1,el2,  ..))=
set('abcd')={'a','b','c',d'}
set({'key1':1,'key2':2})={'key1', 'key2'}
```

N.B. to add a string str1 as a single element use **set**((*str1*,)),
(str1,) is a single element tuple
This is not applicable to a list or dictionary because they are changeable

Set operations
Access element *el* of set *s1:* **for** *el* **in** *s1*
Add element to *s1* = *s1*.**add**(*element*)
Union of two sets = *s1* **|** *s2*
Difference between two sets = *s1* **-** *s2*
Intersection of two set = *s1* **&** *s2*
Test if *s1* is a superset of *s2* = *s1*.**issuperset**(*s2*)
Test if *s1* is a subset of *s2* = *s1*.**issubset**(*s2*)
A set can be used to remove duplicates of a list. e.g. l1=list(set(l1))

11) Iterators Objects

Iterator is an object that goes one by one through a sequence extracting values
Lists, tuples, dictionaries, and sets are all iterable objects
obj_iter **= iter**(*seq1*)= creates an iterator object from sequence *seq1*
next(*obj_iter*) = To get the next element of the iterator object
range(*start*, *stop*, *step*) function that returns a range object with a a sequence of integers from *start* (inclusive) to *stop* (*exclusive*) by *step*.
enumerate(*seq1*) = iterator with index and element of *seq1,*
e.g with list x

```
x=[100,200,300]
iter=enumerate(x)
next(iter)
>>> (0,100)
next(iter)
>>> (1,200)
next(iter)
>>> (3,300)
```

reversed(*seq1*) = creates an iterator from sequence *seq1* in reverse order
zip(*s1*,*s2*) = interleaves two or more sequences into iterable zip object

each element of the zip object is a tuple with elements in s1 and s2 in their order,

e.g.

```
l1=[10,20,30]
l2=[3,4,5]
zip_l=zip(l1,l2)
next(zip_l)
>>>> (10,3)
next(zip_l)
>>>> (20,4)
```

can be used to sum all the elements of two lists, l1 and l2 using list comprehension, e.g.

```
sumlist=[(zl[0]+zl[1]) for zl in zip(l1,l2)]
```

l1=**list**(*it*) Transform Iterator *it* into list *l1*

Generator expression: creates a new generator iterator by processing an iterable one value at time according to a condition

newgen = (*expression* **for** *item* **in** *iterable* **if** *condition* **== True**)

Useful to process one value at time without filling the memory

Generator function: returns one value at time

use **yield** instead of **return**

reminds its state while it is iterated

```
def fgen():
    yield 0
    yield 10
    yield 20
:
it1=fgen()
next(it1)
next(it1)
```

12) File Input/Output

fid=**open**(name, mode) opens file with name

fid= file object

mode="**r**" read, "**w**" write, "**a**" append,

"**x**" write new file (error if file exists),

"**t**" text (default), "**b**" binary (e.g. images)

File Methods:

https://www.w3schools.com/python/python_ref_file.asp

fid.**write**(*string*)

var=fid.**read**(*nbytes*) read n bytes or all the file (default, nbytes=-1)

var=fid.**readline**() read whole line of a file

fid.**close**() close the file (flushes output buffer)

Alternative use **with** (includes open and close

e.g. for reading a file into matrix

```
matrix=[]
with open(file,'r') as f:
    for line in f: # read line by line
    row=[]
    # process content of line
        for item in line.split() # space separated items
            row.append(item)
            # append item as a string to row
    matrix.append(row)
    # append row to matrix
# here file is closed
```

e.g. for writing a file

```
with open(file,'w') as f:
```

```
        f.write(variable)
    # here file is closed
```

13) Modules

https://realpython.com/python-modules-packages/
A module is a file with suffix **.py**
It can contain sections with a function, variable and class
To have code executed only when run as script

```
if __name__ == '__main__':
    .....
```

To use a module

```
import mod1
mod1.fn1() # call function fn1 of the module
```

Define an alias name (usually shorter) for the module, use keyword **as**

```
import mod1 as alias_mod1
alias_mod1.fn1()  # call function fn1 of the module
```

Load a specific section of the module
(i..e a function, variable or class)
without having to use the module name
(import symbol table overwriting local names), use keyword **from**

```
from mod1 import sect_mod1/* (import all sections)
# section is function sect_mod1()
sect_mod1()
    OR
# section is variable sect_mod1
b=sect_mod1
    OR
# section is class sect_mod1()
obj1=sect_mod1()
```

Import specific objects into the local symbol table renaming them

```
from <module_name> import sect_mod1 as alias_name
```

dir(*mod1*) list all the objects in them module, e.g.

```
dir(os.path)
```

Common modules
> **math**
> **cmath**
> **random**
> **string**
> os

- **os.getcwd**()= get currentr working directory
- **os.path.abspath**(path) = return absolute path
- **os.listdir**(path=None)= return a list containing the names of the files in the directory.
- **os.path,exists**(path)= Test whether a path exists
- **os.path.isfile**(path) = Test whether a path is a regular file
- **os.path.isdir**(s) = Return true if the pathname refers to an existing directory.
- **os.path.basename**(p)= Returns the final component of a pathname
- **os.path.getsize**(filename) = Return the size of a file

> **profile**
> **pickle+dbm/shelve** (for database management)

14) Packages

https://realpython.com/python-modules-packages/
Container of **modules** (i.e. .py files) all located in a folder
To access a specific module of *pkg* use dot **.** notation or **from**

import *pkg.mod1*
from *pkg* **import** *mod1*

Package initialisation is done by file **__init__.py** in the folder including global variables
Sub-packages can be used (subfolders in the package folder)
To access a specific module, *mod3*, of *sub_pkg*
import *pkg.sub_pkg.mod3*
from *pkg.sub_pkg* **import** *mod3*

15) Classes, Objects and Methods

Objects are mutable
id(*object*) returns the id number of an object
type(*object*) return the type of an object
A **is** B check if A and B are the same object
A **==** B check is A and B are equal,
if class of A and B has equivalence methods defined,
otherwise it uses **is**
isinstance(object , class) = Boolean function if *object* is an instantiation of *class*

Create a class
class *cname*:

Class Inheritance: class *cname2* (child class) inherits methods of class *cname1* (parent class)
class *cname2*(*cname1*)

Create an object of class *cname*: instantiation of class *cname*
obj1=cname()

Assign attributes to an object, using **dot .** notation
obj1.a=...

A method is a function associated with a class:
class *cname*:
def *cmet1*(*params*):

Call of a method
cname.cmet1(*params*)
Or
obj1.cmet1(*params*)

At instantiation of an object, the function **__init__()** of the class *cname* is executed to initialise the object
def __init__(*self , params*) *# self* = object being instantiated

Conversion of object to string representation is done by function **__str__** in the class
def __str_(*self*)

Operator overloading, redefine use of a certain operator for objects of a certain class (e.g. +, - or /)
def __add__(*self , other*)

Annex 1) Properties of basic types

Type	Iterable	Unique	Mutable	Hashable	Ordered

Int, float, complex	N	N	N	Y	N
string	Y	N	N	Y	Y
list	Y	N	Y	N	Y
tuple	Y	N	N	Y	Y
dict	Y	Y	Y	N	Y
set	Y	Y	Y	N	N

Ordered = elements can be accessed by indexing
Hashable = it has a hash value that does not change during its entire lifetime and it can be used as a key for a dictionary or as an element in a set.

Annex 2) Details on % string format operator (old style)

Ref: https://docs.python.org/3/library/stdtypes.html#old-string-formatting

fs%t where fs is the format string and t is the tuple or dictionary containing arguments

fs contains 2 or more characters:

1. **%** specifies position of value within the tuple providing arguments
2. (*name_value*) OPTIONAL
   Mapping key *name_value* for the specifier in 1.
   In this case value is a dictionary {"*name_value*":value}
3. Conversion flag OPTIONAL
   see table

Flag	Meaning
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a "space" flag).

4. Minimum field width OPTIONAL,
   if = * reads width from next element in tuple *values*
5. Precisions .prec, OPTIONAL,
   if = * reads precisions from next element in tuple *values*
6. Length modifier OPTIONAL
7. Conversion type (see table)

Conversion Type	Meaning
'd'	Signed integer decimal.
'i'	Signed integer decimal.
'o'	Signed octal value.
'x'	Signed hexadecimal (lowercase).
'X'	Signed hexadecimal (uppercase).
'e'	Floating point exponential format (lowercase).

'E'	Floating point exponential format (uppercase).
'f'	Floating point decimal format.
'F'	Floating point decimal format.
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
'c'	Single character (accepts integer or single character string).
'r'	String (converts any Python object using repr()).
's'	String (converts any Python object using str()).
'a'	String (converts any Python object using ascii()).
'%'	No argument is converted, results in a '%' character in the result.

Examples using optional dictionary keys in the formatting string

```
fs="string is: %(name)s"
fs%{'name':'Abcd'}
>>>>> string is: Abcd

fs="Integer number %(int_val)5d\nReal number, %(dec_val)6.4g\na String, %(str_val)10s\n"
values={"int_val":5,"dec_val":0.25, "str_val":'HOUSE'} # dictionary
print(fs%values)
>>>Integer number     5
>>>       Real number,  0.25
>>>       a String,      HOUSE
```

Annex 3) Procedures for specific applications

See folder Basic_Procedures

- **Matrix Initialisation:** matrix_initialisation .py
- **Load and convert a space separated file:** load-convert_file.py

Information for processing Images

iv = index of vertical axis from 0 to nv-1
nv = len(img_in)
ih = index of horizontal axis from 0 to nh-1
nh = len(img_in[0])
img_in[0][0] =>top left pixel