SLICING

--> It is a phenomenon of extracting part of values of a collection.

Syntax : var [SI : EI :UP]

where SI --> Starting Index
EI --> Ending Index (It will be Excluded)
UP --> Updation

--> Slicing works based on indexing and it is possible only on collection which supports indexing

--> If we are extracting from left to right then updation will be positive , else it will be negative

Example :

```
s='mango'
s[0:3:1]
'man'
s[0:2:1]
'ma'
```

Simplified forms :-

```
s='mango'

s[-1:-6:-1]
'ognam'
len(s)
5
s[::-1]
'ognam'


s[0:5:2]
'mno'
s[::2]
'mno'

s='dark'

s[1:4:2]
'ak'
len(s)
4
s[1::2]
'ak'
```

Slicing on list :

```
L[0:3:1]
[10, 2.5, 'Rishab pant']

L[:3]
[10, 2.5, 'Rishab pant']

L[::-1]
[777, (1+2j), 'Rishab pant', 2.5, 10]

L[2][7:11]
'pant'

L=[10,2.5,[99,150,['kingkohli',18],45],'KLRAHUL']
L[2][2][0][:4]
'king'
```

NOTE :

1) Slicing on tuple works exactly like list
2) Slicing on set is not possible because indexing is not possible on set.

Slicing on dictionary :

NOTE:

Slicing on dictionary is not possible directly , but when the values are of collection datatype which supports indexing , then slicing can be applied.

Ex:01)

```
d={10:20,'hi':4.5,7:'good boy'}

d[7][5:8]
'boy'

d['hi'][::-1]  ------>ERROR
```

EX:02)

```
d={'name':'Qspiders','dob':2004,'branch':['rajajinagar','btm']}

d['branch'][0][6:]
'nagar'
d['dob'][:2]  ------>ERROR
```

Slicing → extracting part of values

indexing: Var[index]

s = 'Mango'

'Man' ← s[-2]

Syntax:

Var [SI : EI : UP] (excluded 'g') SI <= <EI

ex: s = 'Mango' (0 1 2 3 4)

Man (0 1 2) ← diff (UP)

s[0 : 3 : 1]

'Man'

Var[SI : EI : UP]

extracting values from Left to Right	extracting values from Right to Left
Up → +ve (+ve indexing)	Up → -ve (-ve indexing)

Simplified forms

1) if SI = 0 or -1
Var[:EI:UP]

2) if EI = len(col) (or) -len(col)-1
Var[SI::UP]

3) if UP = 1
Var[SI:EI]

General Syntax

1) To reverse collection
Var[::-1]

2) To get values present @ even index
Var[::2]

3) To get values present @ odd index
Var[1::2]

s = 'Morning' (0 1 2 3 4 5 6)

Values present @ even index

'Nrig'

Var[SI:EI:UP]

s[0:7:2]

s = 'kerala' (0 1 2 3 4 5)

s[1:6:2] or s[1::2]

SI → 1
EI → 6
UP → 2

Values @ odd index

s = 'Mango' (-5 -4 -3 -2 -1 / 0 1 2 3 4)

L ← R Up → -ve

Var[SI:EI:Up]

s[-1:-6:-1]

(-1) (-2) (-3) (-4) (-5) (-6)

s[-1] → ognam
s[-2]
s[-3]
s[-4] s[-5]

len(s) = 5
-len(ul)-1 = -6

Slicing on List

L = [10, 2.5, 'hi', 9] (0 1 2 3)

L[0:3:1]

or

L[:3]

a = ['Bngalure', 7, 1.5, 'ok'] ('Bngalure': 0 1 2 3 4 5 6 7 8 9)

'luru'

Nested col →

Var[SI:EI:UP]

a[0][5:9:1]

Slicing on dict

Only in 1 case, it is possible

if values are of col D.T which supports indexing

then only slicing is possible

Typecasting

def^n :

int → float

10 → 10.0

Syntax :

dest_var = dest_type (var/val)

Ex: a = 10
b = float (a)
b → 10.0

source_type	dest_type
int	int → No use
	float → add .0
	complex → add 0j
	bool → T/F
	str → stores within Quotes
	list, tuple, set, dict → ⊗ Not possible

source_type	dest_type
float	int → inter part
	float → No use
	complex → add 0j
	bool → T/F
	str → stores within Quotes
	list, tuple, set, dict → ⊗ Not possible

10 → 10+0j
2.5 → 2.5+0j

(a) ± bj → a: int/float; a: RP, b: IP

source_type	dest_type
complex	int, float → ⊗
	complex → No use
	bool → T/F
	str → stores within Quotes
	list, tuple, set, dict → ⊗

source_type	dest_type (T / F)
bool	int → 1 or 0
	float → 1.0 or 0.0
	complex → 1+0j or 0j
	bool → T or F
	str → stores within Quotes
	list, tuple, set, dict → ⊗ Not possible

source_type	dest_type
str	int → string should have only integers
	float → string should have integers with or without decimal points
	complex → —— , ——
	bool → T/F
	str → stores within Quotes
	list, tuple, set → individual values will be stored as separate values within [ ] / ( ) / { }
	dict → ⊗ Not possible

var = 'val1 val2 .... valn'

s = 'okay'

list(s)

↳ ['o', 'k', 'a', 'y']

source_type	dest_type
list	int, float, complex → ⊗ Not possible
	bool → T/F
	str → stores within Quotes
	list → No use
	tuple, set → ( ), { }
	dict → ⊗ Not possible

source_type	dest_type
dict	int, float, complex → ⊗ Not possible
	bool → T/F
	str → stores within Quotes
	list, tuple, set → Give only keys
	dict → No use

NOTE :

CONVERSION OF tuple and set into all datatypes works similar to list.

TYPECASTING

It is a phenomenon of converting type of data from one type to another type.

Syntax : dest_var = dest_type(var/val)

1)int

Ex : a=10
b=int(a)
b
10
c=float(a)
c
10.0
d=complex(a)
d
(10+0j)
e=bool(a)
e
True
f=str(a)
f
'10'
g=list(a) -----> Error
h=tuple(a) -----> Error
i=set(a) -----> Error
j=dict(a) -----> Error

2) float

Ex : a=2.5

int(a)
2
float(a)
2.5
complex(a)
(2.5+0j)
bool(a)
True
str(a)
'2.5'

list(a) ----> Error
tuple(a) ----> Error
set(a) ----> Error
dict(a) ----> Error

3)complex

Ex: a=2+3j
int(a) ----> Error
float(a) ----> Error

complex(a)
(2+3j)

bool(a)
True

str(a)
'(2+3j)'

list(a) ----> Error
tuple(a) ----> Error
set(a) ----> Error
dict(a) ----> Error

4)bool

Ex : a=True
  b=False
int(a)
1
int(b)
0
float(a)
1.0
float(b)
0.0
complex(a)
(1+0j)
complex(b)
0j
bool(a)
True
bool(b)

False
str(a)
'True'
str(b)
'False'

list(a) ----> Error
tuple(a) ----> Error
set(a) ----> Error
dict(a) ----> Error

5) string

s='mango'
a='1234'
b='4.5'

int(s) ------>Error
int(a)
1234
int(b) ------>Error

float(s) ------>Error
float(a)
1234.0
float(b)
4.5

complex(s) ------>Error
complex(a)
(1234+0j)
complex(b)
(4.5+0j)

bool(s)
True
bool('')
False

str(s)
'mango'

list(s)
['m', 'a', 'n', 'g', 'o']

tuple(s)
('m', 'a', 'n', 'g', 'o')

set(s)
{'m', 'g', 'a', 'n', 'o'}

dict(s) ------>Error

6) list

L=[1,2,'hi',2.5]

int(L) ------> Error
float(L) ------> Error
complex(L) ------> Error

bool(L)
True
str(L)
"[1, 2, 'hi', 2.5]"
list(L)
[1, 2, 'hi', 2.5]
tuple(L)
(1, 2, 'hi', 2.5)
set(L)
{1, 2, 2.5, 'hi'}

dict(L) ------> Error

x=['ok',[1,2],(4.5,25)]
dict(x)
{'o': 'k', 1: 2, 4.5: 25}

NOTE : CONVERSION OF TUPLE , SET INTO ALL DATATYPES WORKS SIMILAR TO LIST .

9) dict

d={10:20,2.35:'hi','ok':'deepavali'}

int(d) ------> Error
float(d) ------> Error
complex(d) ------> Error

```
bool(d)
True

str(d)
"{10: 20, 2.35: 'hi', 'ok': 'deepavali'}"

list(d)
[10, 2.35, 'ok']

tuple(d)
(10, 2.35, 'ok')

set(d)
{'ok', 10, 2.35}

dict(d)
{10: 20, 2.35: 'hi', 'ok': 'deepavali'}


examples:-->

complex(7)
(7+0j)

int('1230')
1230
dict({'77'})
{'7': '7'}
dict({(7,7)})
{7: 7}
str(False)
'False'
complex(False)
0j
len(tuple({'5555'}))
1
len({'7777'})
1
len({{7,7,7,7}})  ----error
complex('False')   ----error
dict({(77)})  ----error
int(7+0j)v ----error
int('123.0')  ----error
```

Copy Operation

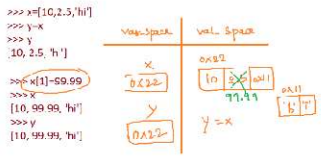
→ Coping context (value) from one var to another

3 types → 1) General / Normal copy
2) shallow copy
3) deep copy

1) General / Normal copy

→ to the same memory location

Syn:
```
next_var = source_var
```
Ex:




```
>>> x=[10,2.5,'hi']
>>> y=x
>>> y
[10, 2.5, 'h']
>>> x[1]=99.99
>>> x
[10, 99.99, 'hi']
>>> y
[10, 99.99, 'hi']
```

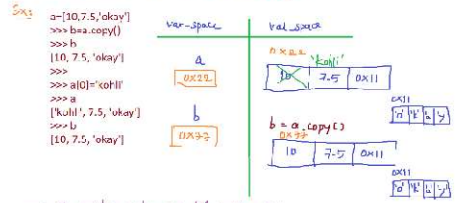
EXAMPLES:

```
a=10
>>> b=a
>>> b
10
>>> id(a)
2267703634512
>>> id(b)
2267703634512
>>>
>>> x=[10,2.5,'hi']
>>> y=x
>>> y
[10, 2.5, 'h']
>>> id(x)
2267741076160
>>> id(y)
2267741076160
>>>
>>> x[1]=99.99
>>> x
[10, 99.99, 'hi']
>>> y
[10, 99.99, 'hi']
```

2) Shallow copy

→ to different memory location

Syntax:
```
dest_var = source_var.copy()
```
Ex:




```
a=[10,7.5,'okay']
>>> b=a.copy()
>>> b
[10, 7.5, 'okay']
>>>
>>> a[0]='kohli'
>>> a
['kohli', 7.5, 'okay']
>>> b
[10, 7.5, 'okay']
```

→ it works only on Mutable col

→ for nested co. it will copy to same memory location (Drawback)

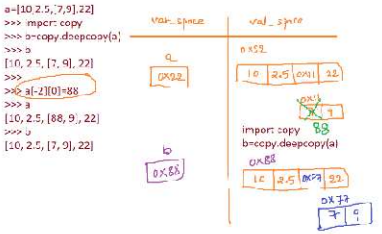
EXAMPLES:

```
a=[10,7.5,'okay']
>>> b=a.copy()
>>> b
[10, 7.5, 'okay']
>>>
>>> a[0]='kohli'
>>> a
['kohli', 7.5, 'okay']
>>> b
[10, 7.5, 'okay']
>>>
>>> id(a[-1])
2048314416176
>>> id(b[-1])
2048314416176
>>>
>>> x=[5,2.2,[9,77]]
>>> y=x.copy()
>>> y
[5, 2.2, [9, 77]]
>>>
>>> x[0]=333
>>> x
[333, 2.2, [9, 77]]
>>> y
[5, 2.2, [9, 77]]
>>>
>>> x[-1][-1]=1.23
>>> x
[333, 2.2, [9, 1.23]]
>>> y
[5, 2.2, [9, 1.23]]
>>>
>>> id(x[-1])
2048314250816
>>> id(y[-1])
2048314250816
```

3) deep copy

→ to different memory location (for nested col also)

Syntax:
```
import copy
dest_var = copy.deepcopy(source_var)
```



EXAMPLE:

```
a=[10,2.5,[7,9],22]
>>> import copy
>>> b=copy.deepcopy(a)
>>> b
[10, 2.5, [7, 9], 22]
>>>
>>> a[-2][0]=88
>>> a
[10, 2.5, [88, 9], 22]
>>> b
[10, 2.5, [7, 9], 22]
>>>
>>> id(a)
3105229159168
>>> id(b)
3105229182208
>>>
>>> id(a[-2])
3105229181504
>>> id(b[-2])
3105219758848
```

COPY OPERATION

--> It is phenomenon of copying a context , from one variable to another variable.

--> It is of 3 types , they are : 1) General / Normal copy
2) Shallow copy
3) Deep copy

1) General/Normal copy :

--> It is phenomenon of copying a context , from one variable to another variable to the same memory location.

--> When a mutable collection is modified , it effects another variable also (drawback).

--> General copy can be done on all the datatypes, but modification will work only on mutable datatypes.

Syntax : dest_var = source_var

Example :

```
a=10
b=a
b
10

id(a)
140723140119256
id(b)
140723140119256

x=[10,20,30]
y=x
y
[10, 20, 30]

y[0]=777

y
[777, 20, 30]
x
[777, 20, 30]
```

2) Shallow copy :

--> It is phenomenon of copying a context , from one variable to another variable to the different memory location.

Syntax : dest_var = source_var.copy()

NOTE :

1) It works only on mutable datatypes
2) In case of Nested collection , it copies to the same memory location (drawback).
3) If a nested collection is modified with respect to one variable , it will affect other variable also.

Example :

```
x=[1.5,22,[1,2,3]]
y=x.copy()
y
[1.5, 22, [1, 2, 3]]

x[2][0]=777
x
[1.5, 22, [777, 2, 3]]
y
[1.5, 22, [777, 2, 3]]

id(x)
2168249701120
id(y)
2168292643648

id(x[-1])
2168292710848
id(y[-1])
2168292710848
```

3) Deep copy :

--> It is a phenomenon of copying the context from one variable to another variable to the different memory location.

--> If there are nested collection then also, it copies to the different memory location.

Syntax : import copy
dest_var = copy.deepcopy(source_var)

Example :

```
a=[10,20,[7,8]]
import copy
b=copy.deepcopy(a)
b
[10, 20, [7, 8]]

a[2][0]=99.99
a
[10, 20, [99.99, 8]]
b
[10, 20, [7, 8]]

id(a)
1163485900416
id(b)
1163491415552

id(a[2])
1163491348608
id(b[2])
1163447489280
```