These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

Lambda Functions are anonymous function means that the function is without a name. As we already know that the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in Python.

```
store = lambda x: "Even_no" if x%2==0 else "Odd_no"

print(store(10))

#o/p

#Even_no

#Lambda, list compression map,filter reduce

word = 'Thisisgood'


# lambda returns a function object

in_rev = lambda string: string.upper()[::-1]  #in_rev reverses and converts string to upper

print(in_rev(word))

#o/p

#DOOGSISIHT
```

Difference Between Lambda functions and def defined function

```
def cube_of(num):
    return num*num*num


def cube_la(num): return num*num*num


# using function defined

# using def keyword

print("calling a function:", cube_of(6))


# using the lambda function

print("using lambda:", cube_la(6))

#o/p

#calling a function: 216

#using lambda: 216
```

Lambda Function with List Comprehension

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

```
find_even = [lambda arg=x: arg * 10 for x in range(1, 5)]


# iterate on each lambda function

# and invoke the function to get the calculated value

for item in find_even:

  print(item())
```

- #o/p
- #10
- #20
- #30
- #40

Lambda Function with if-else

```
Max = lambda num1,num2 : num1 if(num1 > num2) else num2

print(Max(1, 2))
```

#o/p

#2

Filter()

- filter(function, sequence)
- The filter() function in Python takes in a function and a list as arguments.
- This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True.

Lambda functions can be used along with built-in functions like filter(), map() and reduce()

```
li = [7,1,5,9,21,22,25,44,12]

output = list(filter(lambda x: (x % 2 != 0), li))

print(output)
```

#o/p

#[7, 1, 5, 9, 21, 25]

Map()

- The map() function in Python takes in a function and a list as an argument.
- The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

- r = map(func, seq)
- The first argument func is the name of a function and the second a sequence (e.g. a list) seq. map() applies the function func to all the elements of the sequence seq.
- map() returns an iterator.

#Multiply all elements of a list by 2 using lambda and map() function

Python code to illustrate

map() with lambda()

to get double of a list.

li = [3,7,9,20,70]

modified = list(map(lambda x: x*2, li))

print(modified)

#o/p

#[6, 14, 18, 40, 140]

Cartesian product

- Use itertools.product() to generate Cartesian product of multiple lists
- what is cartesian product?
- The Cartesian product is the set of all combinations of elements from multiple sets.
- Import the itertools module.
- is used to make the results easier to read.
- Pass two lists as arguments. itertools.product() returns an object of type
- itertools.product. itertools.product is an iterator,
- so the contents is not output by print()

import itertools

l1 = ['a', 'b', 'c']

l2 = ['X', 'Y', 'Z']

p = itertools.product(l1, l2)

for v in p:

  print(v)

- #o/p
- # ('a', 'X')
- # ('a', 'Y')

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

- # ('a', 'Z')
- # ('b', 'X')
- # ('b', 'Y')
- # ('b', 'Z')
- # ('c', 'X')
- # ('c', 'Y')
- # ('c', 'Z')

#or

```
for v1, v2 in p:
    print(v1, v2)
```

#o/p:

a X

a Y

a Z

b X

b Y

b Z

c X

c Y

c Z

```
from itertools import permutations
txt = ["".join(i) for i in permutations('SKIN')]
print(txt)
```

#o/p

#['SKIN', 'SKNI', 'SIKN', 'SINK', 'SNKI', 'SNIK', 'KSIN', 'KSNI', 'KISN', 'KINS', 'KNSI', 'KNIS', 'ISKN', 'ISNK', 'IKSN', 'IKNS', 'INSK', 'INKS', 'NSKI', 'NSIK', 'NKSI', 'NKIS', 'NISK', 'NIKS']

Execute

- exec is not an expression: a function in Python 3.x.
- #It compiles and immediately evaluates a statement or set of statement contained in a string.
- #Example:

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

```
exec('print(5)')

# prints 5.

# exec 'print 5' nor the exec neither the print is a function there

exec('print(5)\nprint(6)')

# prints 5{newline}6.

exec('if True: print(6)')

# prints 6.

exec('5')

# does nothing and returns nothing.
```

Eval

#eval is a built-in function (not a statement), which evaluates an expression and returns the value that expression produces.

```
#Example:

x = eval('5')

print(x)

# 5

x = eval('%d + 6' % x)

print(x)

# 11

#x = eval('x = 5') # INVALID; assignment is not an expression.

#x = eval('if 1: x = 4') # INVALID; if is a statement, not an expression.
```

enumerate, zip

- enumerate() and zip() are useful when iterating elements of iterable (list, tuple, etc.)
- # in a for loop.

```
names = ['Alice', 'Bob', 'Charlie']

ages = [24, 50, 18]

for i, (name, age) in enumerate(zip(names, ages)):

   print(i, name, age)

# 0 Alice 24

# 1 Bob 50

# 2 Charlie 18
```

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

Copy

- #Assignment statements in Python do not create copies of objects, they only bind names to
- #an object. For immutable objects, that usually doesn't make a difference.
- #But for working with mutable objects or collections of mutable objects, you might be
- #looking for a way to create "real copies" or "clones" of these objects.
- #Essentially, you'll sometimes want copies that you can modify without automatically
- #modifying the original at the same time.
- #A shallow copy means constructing a new collection object and then populating it
- #with references to the child objects found in the original.
- In essence,
- #a shallow copy is only one level deep. The copying process does not recurse and
- #therefore won't create copies of the child objects themselves.
- **Shallow copy**

```
xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(xs)   ##[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ys = list(xs)  # Make a shallow copy
print(ys)  #[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
xs.append(['new sublist'])
print(xs)  #[[1, 2, 3], [4, 5, 6], [7, 8, 9], ['new sublist']]
print(ys)  #[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
xs[1][0] = 'X'
print(xs)
#[[1, 2, 3], ['X', 5, 6], [7, 8, 9], ['new sublist']]
print(ys)
#[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
```

Deep copy

- we (seemingly) only made a change to xs. But it turns out that both sublists at index 1
- in xs and ys were modified.
- Again, this happened because we had only created a shallow copy of the original list.
- Deep copy
- Had we created a deep copy of xs in the first step, both objects would've been

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

- fully independent.
- This is the practical difference between shallow and deep copies of objects.
- A deep copy makes the copying process recursive. It means first constructing a
- new collection object and then recursively populating it with copies of the child objects
- found in the original. Copying an object this way walks the whole object tree to create a
- fully independent clone of the original object and all of its children.

```
import copy

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

print(xs)  #[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

zs = copy.deepcopy(xs)

print(zs)  #[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

xs[1][0] = 'X'

print(xs)  #[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]

print(zs)  #[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

First class functions

- **First class** objects in a language are handled uniformly throughout.
- They may be stored in data structures, passed as arguments, or used in control structures.
- A programming language is said to support first-class functions if it treats functions as first-class objects.
- Python supports the concept of First Class functions.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

1. Functions are objects

Python functions are first class objects.

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

Here we are assigning function to a variable.

This assignment doesn't call the function. It takes the function object referenced by shout and creates a second name pointing to it, yell.

```
# Python program to illustrate functions can be treated as objects

def shout(text):
   return text.upper()


print (shout('Hello'))


yell = shout


print (yell('Hello'))
```

#Output:

HELLO

HELLO

2. Functions can be passed as arguments to other functions:

- Because functions are objects we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called higher-order functions. In the example below, we have created a function greet which takes a function as an argument.
- # Python program to illustrate functions can be passed as arguments to other functions

```
def shout(text):
   return text.upper()

def whisper(text):
   return text.lower()

def greet(func):
   # storing the function in a variable
   greeting = func("""Hi, I am created by a function passed as an argument.""")
   print (greeting)

greet(shout)
greet(whisper)
```

- Output

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, i am created by a function passed as an argument.
```

3. Functions can return another function:

Because functions are objects we can return a function from another function. In the below example, the create_adder function returns adder function.

These notes are only to get basic Knowledge, you'll have to study extensively separately, refer class notes

```
# Python program to illustrate functions. Functions can return another function

def create_adder(x):
   def adder(y):
      return x+y

   return adder

add_15 = create_adder(15)

print (add_15(10))
```

- Output:
  25