Thanks for staying in touch, guys. My name is Akshansh(+91 8384891269, akshanshofficial@gmail.com (mailto:akshanshofficial@gmail.com)). We've been in touch for so long now. It has been 1.5 months together and we have discussed many topics those were distributed as lecture notes. We've discussed 13 lectures.

One of my goals has been to teach you as little python as possible. When there were two ways to do something, I picked one and avoided mentioning the other so that you don't confused.

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary - you can write good code without them -but with them you can sometimes write code that's more concise, readable or efficient and sometimes all three.

Conditional Expression

We saw conditional statements in Lecture notes4 : Conditional Inputs. Conditional statements are often used to choose one of the two values; for example -

In [ ]:

```
if x>0:
    y=math.log(x)
else:
    y=float("nan")
```

This statement checks whether x is positive. If so, it computes math.log. If not, math.log would raise a ValueError. To avoid stopping the program, we generate a "nan", which is a special floating-point value that represents "Not a Number".

We can write this statement more concisely using a conditional expression:

In [ ]:

```
y=math.log(x) if x>0 else float('nan')
```

you can almost read this as normal English. y gets lof(x) if x is greater than 0, otherwise it gets "nan" (not a number)

this is called Lambda funtion or annonymous function. Which makes your code easy to read and concise at the same time.

general syntax for lambda function is like -

expression condition

y=math.log(x) is expression and if x>0 else flaot("nan") is condition. This is as simple as that.

you can use lambda function in functions too like this-

In [7]:

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

calling the function

how factorial works- factorial 4 means= 4x3x2x1

factorial 10 means= 10x9x8x7x6x4x3x2x1

In [9]:

```
factorial(4)
```

Out[9]:

24

we can write it like :

In [10]:

```
def factorial(n):
    return 1 if n==0 else n*factorial(n-1)
```

let's check if it is working properly-

In [11]:

```
factorial(4)
```

Out[11]:

24

you can see that if works perfectly fine.

another use of conditional expressions is handling optional arguments. For example, here is the init method Goodkangaroo

In [12]:

```
def __init__(self,name,contents=None):
    self.name=name
    if contents==None:
        contents=[]
    self.pouch_contents=contents
```

We can rewrite this one like this:

In [13]:

```
def __init__(self,name,contents=None):
    self.name=name
    self.pouch_contents=[] if contents==None else contents
```

see this, how Lambda functions works perfectly in this too.

List Comprehensions

In [22]:

```
def capitalize_all(name):
    new_list=[]
    for letter in name:
        new_list.append(letter.capitalize())
    return new_list
```

In [23]:

```
capitalize_all("akshansh")
```

Out[23]:

```
['A', 'K', 'S', 'H', 'A', 'N', 'S', 'H']
```

We can write this more concisely by using list comprehension like this:

In [24]:

```
def capitalize_all(name):
    return [letter.capitalize() for letter in name]
```

let's check

In [25]:

```
capitalize_all("akshansh")
```

Out[25]:

```
['A', 'K', 'S', 'H', 'A', 'N', 'S', 'H']
```

see that's working perfectly too. This is called list comprehension where you can write enhance the list and make it more concise in terms of readability. It is again lambda in list and it goes like [expression condition]

generator expressions

generator expressions are similar to list comprehensions, but eith parentheses insteadnof square brackets:

In [26]:

```
g= (num**2 for num in range(5))
```

In [28]:

```
print(g)
```

```
<generator object <genexpr> at 0x7fa300a56410>
```

The resukt is a generator object that knows how to iterate through a sewuence of values. Nut unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built in functin next gets the next value from the generator.

In [29]:

```
next(g)  #this will print square of 0
```

Out[29]:

```
0
```

In [30]:

```
next(g)  #prints square of 1
```

Out[30]:

```
1
```

In [31]:

```
next(g)  #prints square of 2
```

Out[31]:

```
4
```

In [32]:

```
next(g)  #prints square of 3
```

Out[32]:

```
9
```

In [33]:

```
next(g)  #prints square of 4
```

Out[33]:

```
16
```

In [34]:

```
next(g)  #our range is (5) that means from 0 to 4, so it will be error
```

```
---------------------------------------------------------------------
-----
StopIteration                             Traceback (most recent call
 last)
<ipython-input-34-e734f8aca5ac> in <module>
----> 1 next(g)

StopIteration:
```

to know all the values in g, you can go like -

In [38]:

```
g= (num**2 for num in range(5))
for value in g:
    print(value)
```

```
0
1
4
9
16
```

Generator expressions are often used with functions like sum max and min

In [40]:

```
sum(num**2 for num in range(5))
```

Out[40]:

```
30
```

In [41]:

```
max(num**2 for num in range(5))
```

Out[41]:

```
16
```

In [42]:

```
min(num**2 for num in range(5))
```

Out[42]:

```
0
```

any and all

python provides a built in function any that takes a sequence of boolean values and returns True if any of the values are True. It works on list:

In [43]:
```
any([False,False,True])
```
Out[43]:
```
True
```

In [44]:
```
any([False,False])
```
Out[44]:
```
False
```

In [45]:
```
any([True,True])
```
Out[45]:
```
True
```

That example isn't very useful because it does the same thing in operator. But we could use any to rewrite some of the search functions we wrote "Search". For example,

In [46]:
```
def avoids(word, forbidden):
    return not any (letter in forbidden for letter in word)
```

In [49]:
```
avoids("akshansh", "z")
```
Out[49]:
```
True
```

see z is not in akshansh, returns true

In [50]:
```
avoids("akshansh", "a")
```
Out[50]:
```
False
```

"a" is in akshansh, returns false.

SETS

In [61]:

```
def finding(d1,d2):
    response=dict()
    for key in d1:
        if key not in d2:
            response[key]=None
    return response
```

you have to pass two arguments, in two sets and it will check what is in d1 that is not in d2. If something is in d1 and not in d2 it will assign None to that and store it in a dictionary.

In [62]:

```
finding((4,5,6,7,8),(3,4,6,11,56))
```

Out[62]:

```
{5: None, 7: None, 8: None}
```

Python provides another built in type called a set that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operatins.

In [64]:

```
def finding(d1,d2):
    return set(d1)-set(d2)
```

In [65]:

```
finding((4,5,6,7,8), (3,4,6,11,56))
```

Out[65]:

```
{5, 7, 8}
```

see 5,7,8 were not in d2 so set function created a dictionary of those.

don't be confused, it looks like a dictionary, but it is set. it also stores in {}

How can we check duplicates

In [71]:

```
def checking_duplicates(Input):
    new_dict ={}
    for element in Input:
        if  element in new_dict:
            return True
        new_dict[element]=True
    return False
```

checking in tuple

g p

In [72]:

```
checking_duplicates((2,3,4,4))
```

Out[72]:

True

In [73]:

```
checking_duplicates((1,2,3,4,5,6))
```

Out[73]:

False

check in a list

In [74]:

```
checking_duplicates([1,2,3,4,4,5])
```

Out[74]:

True

check in a dictionary

In [70]:

```
checking_duplicates({2:4,3:9,4:16})
```

Out[70]:

False

let's use some fraction of brain and use lambda to concise this

In [75]:

```
def checking_duplicate(Input):
    return len(set(Input))- len(Input)
```

In [76]:

```
checking_duplicate([1,2,3,4])  #no duplicate returns 0
```

Out[76]:

0

In [77]:

```
checking_duplicate([1,2,3,3,4])  #1 duplicate returns -1
```

Out[77]:

```
-1
```

In [78]:

```
checking_duplicate([1,2,2,3,3,4,4])  #3 duplicates returns -3
```

Out[78]:

```
-3
```

counters

counter is like a set, except that if an element appears more than once. The counter keeps track of how mant times it appears. If you are familiar with the mathematical idea of a multiset, a counter is a natural way to represent a multiset. Anyway leave it, just focus on the code-

counter is defined in a standard module called collections, so you have to import it. You can initialize a Counter with a string, list or anything else that supports iteration:

In [80]:

```
from collections import Counter
count=Counter("akshansh")
count
```

Out[80]:

```
Counter({'a': 2, 'k': 1, 's': 2, 'h': 2, 'n': 1})
```

see it counted how many time a letter appears in my name.

In [83]:

```
count['d'] #prints 0 becuase no 'd' in my name
```

Out[83]:

```
0
```

gathering keyword args

You havn't thought what if we want to pass multiple argument in a function. How to do that?

use *args to pass many argument in a tuple form

In [84]:

```
def takingMultiple(*args):
    print(args)
```

In [85]:

```
takingMultiple(1,2,4.0,5)
```

```
(1, 2, 4.0, 5)
```

see *agrs, return a tuple

what if you want to gather keyword argument:

In [86]:

```
takingMultiple(1,2,3, fourth='4')
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-86-04ab6091a8fd> in <module>
----> 1 takingMultiple(1,2,3, fourth='4')

TypeError: takingMultiple() got an unexpected keyword argument 'fourt
h'
```

for taking keyword argument, use ** kwargs

In [87]:

```
def Taking_multiple(*args,**kwargs):
    print(args,kwargs)
```

In [88]:

```
Taking_multiple(1,2,3, fourht='4')
```

```
(1, 2, 3) {'fourht': '4'}
```

notice * saves arguments(args) in a tuple and ** saves keywords arguments (kwargs) in a dictionary where first is treated as key and second as value.

Thank me later, this lecture will be quite useful for you. My time is up for this lecture. Stay safe and stay at Home. Have a great day.

In [ ]:

```

```

Akshanshofficial@gmail.com