Python_notes

Python is an interpreted, high-level, general-purpose programming language.

Python language is case sensitive, it can also be used for scripting.

Comments:

Single line comment (#)

Eg: **#**Hey I am learning python

Multiline comment (""" """)

Eg: **"""** I am learning python, one could use multi line comments to add notes in between my code to understand logic at a later point in time. During runtime comments are ignored. **"""**

White spaces or INDENTATION

Indentation will be used in functions, loops to create blocks or give structure to code. White Spaces would rise error, if you use them unnecessarily.

Print statements

input	output
i=10 print(i) print("i=",i) print("i=%d"%i)	10 i= 10 i=10
i,j = 10, 15 print('output :i=%s,j=%s'%(i,j))	output :i=10,j=15
name = "Mary" age = 32 stuff_in_string = "Shepherd {} is {} years old.".format(name, age) print(stuff_in_string)	Shepherd Mary is 32 years old.
name,age = "Mary", 32 print("Shepherd {} is {} years old.".format(name, age))	Shepherd Mary is 32 years old.

Objects & Data Structure types:

Numbers:

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number.

Examples	Number type
1,2,-5,1000	Integers
1.2,-0.5,2e2,3E2	Floating-point numbers

Variable Naming convention:

Using variable names can be a very useful way to keep track of different variables in Python. The names you use when creating these labels need to follow a few rules:

1. Names cannot start with a number.
2. There can be no spaces in the name, use _ instead.
3. Can't use any of these symbols :'",<>/?|\()!@#$%^&*~-+
3. It's considered best practice (PEP8) that the names are lowercase.

Variable Assignments:

```
# Let's create an object called "a" and assign it the number 5
a = 5

# Adding the objects
a+a

# Use object names to keep better track of what's going on in your code!
my_income = 100
tax_rate = 0.1
my_taxes = my_income*tax_rate
```

Strings:

Uses sequences in Python. Uses " " or ' '

Sequences are a collection of objects that are stored in an *order*. *Retrieved by index.*

A String is immutable

Lists:

Uses sequences in Python. Uses [ ] with ,

Sequences are a collection of objects that are stored in an *order*. *Retrieved by index.*

A list is mutable

```
# Assign a list to an variable named my_list
my_list = [1,2,3]

my_list = ['A string',23,100.232,'o']
```

A list can contain objects of different types.

Dictionaries:

key : value notation/pair. Uses { } with ,
Uses mappings in Python. Similar to hash in ruby or objects in javascript.
Mappings are a collection of objects that are stored by a *key*. *Retrieved by key*
A Dictionary is mutable, *dictionaries cannot be sorted as they are mappings* not a sequence.

```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}

#Dictionaries are very flexible in the data types they can hold
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

#Can then even call methods on that value
my_dict['key3'][0].upper()

#We can also create keys by assignment
# Create a new dictionary
d = {}
d['animal'] = 'Dog'
```

A few Dictionary Methods

```
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}

# Method to return a list of all keys
d.keys()

# Method to return a list of all values
d.values()
```

Nesting with Dictionaries

```
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

```
# Keep calling the keys
d['key1']['nestkey']['subnestkey']

# Method to return tuples of all items  (we'll learn about tuples soon)
d.items()
```
o/p: [('key3', 3), ('key2', 2), ('key1', 1)]

Tuples:

Uses sequences in Python. Uses ( ) with ,

Sequences are a collection of objects that are stored in an *order*. *Retrieved by index.*

A Tuple is immutable

"Tuples are like lists with the major distinction being that tuples are immutable."

```
# Can create a tuple & check len just like a list
t = (1,2,3)
len(t)

# Can also mix object types
t = ('one',2)

# Slicing just like a list & use indexing
t[-1] #o/p: 2
t[0]
```

Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Two of them are:

```
# Use .index to enter a value and return the index
t.index('one')

# Use .count to count the number of times a value appears
t.count('one')
```

When to use Tuples

Used when immutability is necessary. It provides a convenient source of data integrity.

Files: refer other sources

Set and Booleans

Sets are an unordered collection of *unique* elements. We can

```
#construct them by using the set() function.
x = set()
```

 # We add to sets with the add() method
x.add(1)
x.add(2) #add() takes exactly one argument
o/p: {1,2}

Note: the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

 # Create a list with repeats
l = [1,1,2,2,3,4,5,6,1,1]

 # Cast as set to get unique values
set(l)
o/p: {1, 2, 3, 4, 5, 6}

A set is only concerned with unique elements!

Booleans

Python comes with Booleans (with predefined **True** and **False** displays that are basically just the integers **1** and **0**). It also has a placeholder object called **None**.

 # Set object to be a boolean
a = True
 # Output is boolean
1 > 2

We can use None as a placeholder for an object that we don't want to reassign yet:

 # None placeholder
b = None

Continuation character ('\')

This character is used to continue a line to the next line.

Doesn't work	Works
Eg: print "Python" "Course" **o/p:** Python	**Eg:** print "Python"\ "Course" **o/p:** Python Course
Eg: print 7 -4 **o/p:** 7	**Eg:** print 7\ -4 **o/p:** 3

Operators:

Comparison operators:

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(5 == 5) is true.
!=	If values of two operands are not equal, then condition becomes true.	(5 != 6) is true
<>	If values of two operands are not equal, then condition becomes true.	(5 <> 6) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(2 > 3) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(2 < 3) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(2 >= 3) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(2 <= 3) is true.

Chained comparison operators:

Ability to *chain* multiple comparisons to perform a more complex test.

Input condition	Input condition (2nd way)	Output
1 < 2 < 3	1<2 and 2<3	true
1 == 2 < 3	1==2 or 2<3	true

Statements:

Python vs Other Languages

Let's create a simple statement that says: **"If a is greater than b, assign 2 to a and 4 to b"** Take a look at these two if statements (we will learn about building out if statements soon).

Version 1 (Other Languages)

```
if (a>b){
    a = 2;
    b = 4;
}
```

Version 2 (Python)

```
if a>b:
    a = 2
    b = 4
```

You'll notice that Python is less cluttered and much more readable than the first version. How does Python manage this?

Let's walk through the main differences:

Python gets rid of () and {} by incorporating two main factors: **a *colon*** and ***whitespace*.** The statement is ended with a colon, and whitespace is used (indentation) to describe what takes place in case of the statement.

Another major difference is the lack of semicolons in Python. Semicolons are used to denote statement endings in many other languages, but in Python, the end of a line is the same as the end of a statement.

Indentation:

Here is some pseudo-code to indicate the use of whitespace and indentation in Python:

```
if x:
   if y:
      code-statement
else:
   another-code-statement
```

Note: Python is so heavily driven by code indentation and whitespace. This means that code readability is a core part of the design of the Python language.

If else statements:

```
x = False

if x:
   print 'x was True!'
else:
   print 'I will be printed in any case where x is not true'
```

elif else statements:

```
loc = 'Bank'

if loc == 'Auto Shop':
   print 'Welcome to the Auto Shop!'
elif loc == 'Bank':
   print 'Welcome to the bank!'
else:
   print "Where are you?"
```

Indentation very important !!!!!!

For Loops

A for loop acts as an iterator in Python, it goes through items that are in a *sequence* or any other iterable item. Objects that we've learned about that we can iterate over include strings,lists,tuples, and even built in iterables for dictionaries, such as the keys or values.

Here's the general format for a for loop in Python:

```
for item in object:
   statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use if statements to perform checks.

Let's go ahead and work through several example of for loops using a variety of data object types. we'll start simple and build more complexity later on.

##Example 1 Iterating through a list.
We'll learn how to automate this sort of list in the next lecture

```
numbers = [1,2,3,4,5,6,7,8,9,10]
        for num in numbers:
            print num
```

Example 2

Let's now look at how a for loop can be used with a string:

```
        for letter in 'This is a string.':
            print letter
```

Example 3

Let's now look at how a for loop can be used with a tuple:

```
        tup = (1,2,3,4,5)
        for t in tup:
            print t
        l = [(2,4),(6,8),(10,12)]
        for tup in l:
            print tup
        # Now with unpacking!
        for (t1,t2) in l:
            print t1

        #dictionaries using for loop
        d = {'k1':1,'k2':2,'k3':3}
        for item in d:
            print item
```

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

```
        for k,v in d.items():
            print(k)
            print(v)
```

items() now return iterators

While loops

The while statement in Python is one of most general ways to perform iteration. A while statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statement
else:
    final code statements
```

#example while loop

```
x = 0

while x < 10:
   print('x is currently: ',x)
   #print(' x is still less than 10, adding 1 to x')
   x+=1
else:
   print('All Done!')
```

o/p:

```
x is currently:  0
x is currently:  1
x is currently:  2
x is currently:  3
x is currently:  4
x is currently:  5
x is currently:  6
x is currently:  7
x is currently:  8
x is currently:  9
All Done!
```

break, continue, pass:

We can use break, continue, and pass statements in our loops to add additional functionality for various cases. The three statements are defined by:

break: Breaks out of the current innermost loop.
continue: Goes to the top of the innermost loop.
pass: Does nothing at all.

break and continue statements can appear anywhere inside the loop's body,but we will usually put them further nested in conjunction with an if statement to perform an action based on some condition.

#example for a while loop with continue

```
x = 0

while x < 10:
   print 'x is currently: ',x
   print ' x is still less than 10, adding 1 to x'
   x+=1
   if x ==3:
      print 'x==3'
   else:
```

print 'continuing...'
continue

Range:
This is a generator in python 3 which generates till the given range But doesn't store in any variable, makes it ideal for large iterations in for loop like 1 million or so..

#It doesn't store generated range, see the below code
print(range(10))
o/p: range(0,10)
#print numbers from 0 to 4
for num in range(5):
print(num)
o/p: 0
1
2
3
4

Using else statement with Loops:

Python supports to have an else statement associated with a loop statement

- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

```
for num in range(10,20):     #to iterate between 10 to 20
   for i in range(2,num):    #to iterate on the factors of the number
      if num%i == 0:         #to determine the first factor
         j=num/i             #to calculate the second factor
         print '%d equals %d * %d' % (num,i,j)
         break #to move to the next number, the #first FOR
   else:                  # else part of the loop
      print num, 'is a prime number'
```

```
o/p:
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
```

```
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

List comprehension:

a one line for loop built inside of brackets. Returns a list.

 # Square numbers in range and turn into list

lst = [x**2 for x in range(0,11)]
lst
o/p: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

 # Check for even numbers in a range

lst = [x for x in range(11) if x % 2 == 0]
lst
o/p: [0, 2, 4, 6, 8, 10]

 # Convert Celsius to Fahrenheit

celsius = [0,10,20.1,34.5]
fahrenheit = [((float(9)/5)*temp + 32) for temp in Celsius]
fahrenheit
o/p: [32.0, 50.0, 68.18, 94.1]

#nested list comprehensions

lst = [x**2 for x in [x**2 for x in range(11)]]
lst
o/p: [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]

Functions

They are logical blocks that can be used to create modularity and reusability.

```
def add_num(num1,num2):
    return num1+num2
result = add_num(4,5)
print(add_num('one','two'))
print(result)
```

o/p: 9
onetwo

Methods are Inbuilt functions or functions inside an object or class.

Eg: len("string") => **o/p:** 6

Lambda

Lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

#lambda with parameters

```
adder = lambda x,y : x+y
adder(2,3)
```

o/p: 5

#Reverse a string:

```
rev = lambda s: s[::-1]
rev('hello')
```

o/p: 'olleh'

lambda expressions really shine when used in conjunction with map(),filter() and reduce().

Scope:

```
x = 25 #variable with global scope

def printer():
   x = 50 #variable with local scope
   return x

print(x)
print(printer())
```

o/p: 25
50

LEGB types in scope:

L: Local — Names assigned in any way within a function (def or lambda)), and not declared global in that function.

E: Enclosing function locals — Name in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open,range,SyntaxError,...

Scope of the variable:

Variables declared inside a function are local to the function they are not related in any way to other variables with the same names used outside the function.

#example demonstrating scopes

```
x = 50

def func():
   global x
   print('This function is now using the global x!')
   print('Because of global x is: ', x)
   x = 2
   print('Ran func(), changed global x to', x)
print('Before calling func(), x is: ', x)
func()
print('Value of x (outside of func()) is: ', x)
```

o/p: Before calling func(), x is: 50
This function is now using the global x!
Because of global x is: 50
Ran func(), changed global x to 2
Value of x (outside of func()) is: 2

Object Oriented Programming

EVERYTHING IN PYTHON IS AN OBJECT.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the doc string defined in the function's source code.

What is an object?

Different programming languages define “object” in different ways. In some, it means that all objects must have attributes and methods; in others, it means that all objects are subclassable. In Python, the definition is looser; some objects have neither attributes nor methods and not all objects are subclassable But everything is an object in the sense that it can be assigned to a variable or passed as an argument to a function.

Objects

EVERYTHING IN PYTHON IS AN OBJECT.

```
print( type(1) )
print( type([]) )
print( type(()) )
print( type({}) )
```

o/p: <class 'int'>
<class 'list'>
<class 'tuple'>
<class 'dict'>

All the above are built-in object types, we create custom objects using **class** keyword.

Class is a blueprint or logical grouping
Object is a runtime representation
attribute is a characteristic of an object.
method is an operation we can perform with the object.
__init__() is used to initialize the attributes of an object.

https://jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/

#explaining class with car example

```
class Car(object):
    def __init__(self, model, color, company, speed_limit):
        self.color = color
        self.company = company
        self.speed_limit = speed_limit
        self.model = model
```

```
maruthi_suzuki = Car("ertiga", "black", "suzuki", 60)
audi = Car("A6", "red", "audi", 80)
```

Let's break down what we have above.The special method __init__() is called automatically right after the object has been created:

```
def __init__(self, model, color, company, speed_limit):
```

Each attribute in a class definition begins with a reference to the instance object("self"). The model, color, company, speed_limit are the arguments. The values passed during the class instantiation are assigned to the class attributes.

```
self.color = color
self.company = company
self.speed_limit = speed_limit
self.model = model
```

These attributes can now be accessed as

```
print(audi.model)
print(audi.color)
```

Methods:

#Another example to explain init and methods

Find out the cost of a rectangular field with breadth(b=120), length(l=160). It costs x (2000) rupees per 1 square unit?

```
class Rectangle:
    def __init__(self, length, breadth, unit_cost=0):
        self.length = length
        self.breadth = breadth
        self.unit_cost = unit_cost

    def get_perimeter(self):
        return 2 * (self.length + self.breadth)

    def get_area(self):
        return self.length * self.breadth

    def calculate_cost(self):
        area = self.get_area()
        return area * self.unit_cost
# breadth = 120 cm, length = 160 cm, 1 cm^2 = Rs 2000
r = Rectangle(160, 120, 2000)
```

```
print("Area of Rectangle: %s cm^2" % (r.get_area()))
print("Cost of rectangular field: Rs. %s " %(r.calculate_cost()))
```

Inheritance

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

```
class Animal(object):
    def __init__(self):
        print("Animal created")

    def whoAml(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAml(self):
        print("Dog")

    def bark(self):
        print("Woof!")

d = Dog()
d.whoAml()
d.eat()
d.bark()
```

o/p: Animal created
Dog created
Dog
Eating
Woof!

The derived class inherits the functionality of the base class.

- It is shown by the eat() method.

The derived class modifies existing behavior of the base class.

- shown by the whoAmI() method.

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

Special Methods:

The **__init__()**, **__str__()**, **__len__()** and the **__del__()** methods.

These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

```
class Book(object):
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title:%s , author:%s, pages:%s " %(self.title, self.author, self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
book = Book("Python Rocks!", "Jose Portilla", 159)

#Special Methods
print(book)
print(len(book))
del book
```

o/p: A book is created
Title:Python Rocks! , author:Jose Portilla, pages:159
159
A book is destroyed

You can use **issubclass()** or **isinstance()** functions to check a relationships of two classes and instances.

1. The **issubclass(sub, sup)** boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup.
2. The **isinstance(obj, Class)** boolean function returns true if obj is an instance of class Class or is an instance of a subclass of Class.

Abstract Base Classes: refer other sources

The `abc` module contains a metaclass called `ABCMeta` (metaclasses are a bit outside the scope of this article). Setting a class's metaclass to `ABCMeta` and making one of its methods *virtual* makes it an ABC. A *virtual* method is one that the ABC says must exist in child classes, but doesn't necessarily actually implement.

Static methods are those whose logic doesn't change. So there isn't any need for the argument self.

```
class Car(object):
    @staticmethod
    def make_car_sound():        #no need to pass self
        print('VRooooommmm!')

hi = Car
hi.make_car_sound()
```

o/p: VRooooommmm!

Exception Handling

##try and except

The basic terminology and syntax used to handle errors in Python is the try and except statements. The code which can cause an exception to occur is put in the *try* block and the handling of the exception is the implemented in the *except* block of code. The syntax form is:

```
try:
        You do your operations here...
        ...
except Exception1:
        If there is Exception1, then execute this block.
        ...
except Exception2:
        If there is Exception2, then execute this block.
        ...
else:
        If there is no exception then execute this block.

finally:
        This code block would always be executed.
```

#lets open a file with read permission and try to write to it

```
try:
   f = open('testfile','r')
   f.write('Test write this')
except IOError:
   # This will only check for an IOError exception and then execute this print statement
  print("Error: Could not find file or read data")
else:
  print("Content written successfully")
  f.close()
```

o/p:

Error: Could not find file or read data

Modules & Packages

module:

```
# import the module
import math

# use it (ceiling rounding)
math.ceil(2.4)
```

Exploring built-in modules:

Dir and help are 2 methods used to explore built-in modules.

```
print(dir(math))
```

o/p:

['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

```
help(math.ceil)
```

o/p:

```
Help on built-in function ceil in module math:

ceil(...)
    ceil(x)

    Return the ceiling of x as an int.
    This is the smallest integral value >= x
```

Custom modules & packages:

Writing modules

Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

Writing packages

Packages are name-spaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which MUST contain a special file called ***__init__*.py**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the ***_init_*.py** file inside the foo directory.

To use the module bar, we can import it in two ways:

```
# Just an example, this won't work
import foo.bar

# OR could do it this way
from foo import bar
```

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's name-space.

The ***_init_*.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the ***_all_*** variable

Built-in Functions

All the built-in functions return values are to be type casted into list(or iterables) to use.

Map:

Map takes a function(or lambda) and an iterable object as arguments

#example1

```
a = [1,2,3,4]
b = [5,6,7,8]

d = list(map(lambda x,y:x+y, a,b))
print(d)
```

o/p: [6, 8, 10, 12]

#example2

```
C_temp = [0, 22.5, 40,100]
F_temp = list(map(lambda t: (float(9)/5)*t + 32, C_temp))

print(F_temp)
```

o/p: [32.0, 72.5, 104.0, 212.0]

Filter:

The function filter(function(),l) needs a function as its 1st argument & iterable as 2nd argument.
The function needs to return a Boolean value (either True or False).

#example

```
num = range(20)
even = list( filter(lambda x: x%2==0, num) )
print(even)
```

o/p: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

Zip:

#zip is defined by the shortest iterable length

```
x = [1,2,3]
y = [4,5,6,7,8]

# Zip the lists together
zip(x,y)
```

o/p: [(1, 4), (2, 5), (3, 6)]

#using dictionary

```
d1 = {'a':1,'b':2}
d2 = {'c':4,'d':5}

zip(d1,d2)
```

o/p: [('a', 'c'), ('b', 'd')]

Simply iterating through the dictionaries will result in just the keys. We would have to call methods to mix keys and values

Task: use zip a to switch the keys and values of the two dictionaries:

```
def switcharoo(d1,d2):
        dout = {}
        for d1key,d2val in zip(d1,d2.values()):
                dout[d1key] = d2val
        return dout
d1 = {'a':1,'b':2}
d2 = {'c':4,'d':5}
print(switcharoo(d1,d2))
```

o/p: {'a': 4, 'b': 5}

Enumerate:

Enumerate allows you to keep a count as you iterate through an object. It does this by returning list of tuples in the form (count,element)

#example1

```
lst = ['a','b','c']
print(list(enumerate(lst)))
print()
for number,item in enumerate(lst):
    print(item," - ",number)
```

o/p:

[(0, 'a'), (1, 'b'), (2, 'c')]

a - 0
b - 1
c - 2

#example2

```
for count,item in enumerate(lst):
    if count >= 2:
        break
    else:
        print item
```

o/p: a
b

All & Any

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable.

1. all() will return True if all elements in an iterable are True.

2. any() will return True if any of the elements in the iterable are True.

#example
lst = [True,True,False,True]
all(lst) #Returns False because not all elements are True.
any(lst) #Returns True because at least one of the elements in the list is True

Complex:

complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.
Useful while doing math or engineering that requires complex numbers (such as dynamics,control systems, or impedance of a circuit).

#Create 2+3j
complex(2,3)

#We can also pass strings:
complex('12+2j')