Python-in-8 pages

Please visit my career site for all examples: https://biddu7.github.io/

1. Basics: A. Datatypes & Conversion

SL	Category / Datatype	Variable declaration	Class	Type Conversion
1	Numeric – **int, float, complex**	v1 = 1 # *int* | v2 = 1.0 # *float*	type(v1) <class 'int'>| type(v2) <class 'float'>	**int() | float()**
2	Text – **String**	v3 = "apple"	type(v3) is <class 'str'>	**str()**
3	Sequence – **List**	v4 = ['apple','banana','cucumber']	type(v4) is <class 'list'>	**list()**
4	Sequence – **Tuple**	v5 = ('apple','banana','cucumber')	type(v5) is <class 'tuple'>	**tuple()**
5	Set – **Set, Frozenset**	v6a={'apple','banana'}|v6b=set((1,2,3))	type(v6a | vb6) is <class 'set'>	**set()**
6	Mapping – **Dictionary**	v7 = {'apple':1,'banana':2,}	type(v7) is <class 'dict'>	**dict()**
7	Boolean – **bool**	v8 = True	type(True) is <class 'bool'>	**bool()**
8	Others – (Binary, None)	(Binary) **bytes, bytearray, memoryview**, (None type) **None**		

Type Conversions	Example	Note
Data Type Checking	print(True if **type([var4])==list** else False) >> True	same for all types – **int, str, set, dict, tuple, float.bool**
Float to Int | Int to Float	**int(1.122)** >> 1 | **float(1)** >> 1.0 | **format(1, '.3f')** >> 1.000	**type(format(1, '.3f')) >> str** (and not float)
int/float to string	**str(1.233)** >> 1.233 | **str(1)** >> 1 | print([str(1)]) >> ['1']	quotes doesn't not appear, unless used in list/set/dict
string to int/float	**float('1.222')** >> 1.222 | **int('1.222')** >> error | **int('1')** >> 1	To convert floating string to int follow **Str->Float->Int**
str to list	**list('Joydeep')** >> ['J', 'o', 'y', 'd', 'e', 'e', 'p']	**list(str)** splits all characters & returns a list of all chars.
str to tuple	**tuple('Joydeep') >> ('J', 'o', 'y', 'd', 'e', 'e', 'p')**	**tuple(str)** returns list of all chars in a tuple
str to set	**set('joydeep basu') >>** {'s', 'y', 'd', ' ', 'u', 'e', 'j', 'p', 'o'}	**set(str)** returns all **Unique char**s as set items.
str to set to list	**list(set('joydeep basu')) >>** ['s', 'y', 'd', ' ', 'u', 'e', 'j', 'p', 'o']	**list(set(str))** returns a **list of unique chars** in string.
Iterables to dictionary (iterable= str/list/tuple)	**1. {k:v for k,v in enumerate**(ite, ***start*=0)}** *(ite: str/list/tuple)* **2. {i:ite[i] for i in range(len(ite))}**	**dict** comprehension on **string/list** returning a dictionary with char **Index**:**value** as **key:value** pair
dict keys/values to list	**list({1:'Joy', 2:'Pa',3:'Tia'})** >> [1, 2, 3] **list({1:'Joy', 2:'Pa',3:'Tia'}.values()) >>** ['Joy', 'Pa', 'Tia']	**list(dict1)** returns all keys as list. **list(dict1.values())** returns all dictionary values as list.

dict keys/values to tuple	**tuple({1:'Joy', 2:'Pa',3:'Tia'})** >> (1, 2, 3)
	tuple({1:'Joy', 2:'Pa',3:'Tia'}.values()) >> ('Joy', 'Pa', 'Tia')
		tuple(dict1) >> all dictionary keys as tuple.
		tuple(dict1.values()) >> all dictionary values as tuple.

B. Operations

Operations	Example	Variable	Note

Addition (+) Subtraction (-)
Multiplication (*) Division(/)
	2 + 2 = 4; 5 - 2 = 3;
	3 * 3 = 9; 22 / 8 = 2.75;
		Increment	**var += 1 >>** var = var + 1

Exponent (**)	**2 ** 3 = 8 | pow(2,3) = 8**	Decrement	**var -= 1 >>** var = var - 1
LCM & GCD (Math module)	arr1: [2, 3, 5, 6] | **math.lcm(*arr1)** >> 30 arr2: [24, 48, 36, 96] | **math.gcd(*arr2)** >> 12	Multiply	**var *= 1 >>** var = var * 1
Integer division (//)	**22 // 8 = 2 | divmod(22,8)[0] = 2**	Divide	**var /= 1 >>** var = var / 1
Modulus/Remainder (%)	**22 % 8 = 6 | divmod(22,8)[1] = 6**	Mod	**var %= 1 >>** var = var % 1
Comparison & logical operators	(Comparison) **==** Equal to, **!=** Not equal to, **<, >**, **<=, >=** (logical) **and, or, not**		

C. Python Built-in Functions

Category	Built-in Functions
Create **new variable** or **Type conversion**	**list(), tuple(), set(), dict(), bool(), float(), int(), str(), type()**
Change **Number system**	**bin(),oct(),hex()** → Converts an **integer number** to a binary/Octal/Hexadecimal (lowercase) **string**.
General - **Common** functions	**print()** – for printing, **input()** – to take user input in string format, **import()** – invoked by import statement, **len()** – measure element length in a iterator, **open()** – open a file (text stream)
Numbers operations (*for **Math** functions like ceil, floor – math library to be imported)	**abs()** - absolute value of a number. | **divmod()**>Returns quotient, remainder tuple | Float number rounding: **round()**, **ceil(), floor()** | **pow()** – x to the power y.| **sum(),max(),min()** – [sum|max|min] of iterable element. | **complex()** - Return a complex number with the value **real**+**imaginary***1j.
String built-in functions	**ord()** Returns **ascii code** of a character (**char->ascii**). Ex. **ord('a') >> 97**; **ord('A') >> 65** **chr()** Returns a **character** from its ascii value (**ascii->char**). Ex. **chr(97) >> a; chr(65) >> A** **ascii()** Return a string > **printable representation** of an object. **ascii('a') >> 'a'; ascii('A') >> 'A'** **format()** Convert a value to a "formatted" representation. Ex. **print(format('1', ".3f"))** >> 1.000 **round()** Return number rounded to ndigits precision after the decimal point. **slice()** Return a sliced object representing a set of indices. Takes 3 params (**start, end, step**).
Module **namespace** **Runtime variables** in scope functions	**dir()** Return the list of names in the current local scope. **id()** Return the "identity" of an object. **globals()** Return the dictionary implementing the current module namespace. **locals()** Update and return a dictionary with the current local symbol table. **vars()** Return the dict attribute for any other object with a dict attribute.
Encoding/Decoding functions	**bytes()** - Return a new "bytes" object. | **bytearray()** - Return a new array of bytes. **hash()** - Return the hash value of the object.
Dynamic execution or evaluate expression	**eval()** - Evaluates and executes an expression. | **exec()** - Dynamic execution of Python code.
Execution & **Debugging** functions	**exit()** - Exits the whole execution when called. | **help()** - Invoke the built-in help system. **breakpoint()** – Pauses code execution & starts debugging **compile()**- Compile the source into a code or AST object.

Class Built-in functions

isinstance() Return True if the object argument is an instance of an object.
issubclass() Return True if class is a subclass of classinfo.
hasattr() True if the string is the name of one of the object's attributes.
(get/set)attr() Return/set the value of the named attribute of object.
delattr() Deletes the named attribute, provided the object allows it.
super() Return a proxy object that delegates method calls to a parent/sibling

repr() Returns printable **string representation** of an object.
object() Return a new featureless object.
callable() Return True if the object argument is callable, False if not.
property() Return a property attribute.
classmethod() Transform a method into a class method.
staticmethod() Transform a method into a static method.

2. Iterables/Collection Datatypes

A. Common Operations

Operation	String	List	Tuple	Set
length count	**len()**	**len()**	**len()**	**len()**
Specific element **count**	<string>.**count**(ele)	<list>.**count**(ele)	<tup>.**count**(ele)	always **1**. (**unique**).
Element position (**index**)	s.**index**(e) | s.**rindex**(e)	l.**index**(e)	t.**index**(e)	NA (**un-ordered**)
Element **Type** determine	is(decimal|digit|alpha..)	is(decimal|digit|alpha)	is(decimal|digit|alpha)	is(decimal|digit|alpha)
Slicing iterable	**[start : end** : **step]**	**[start : end** : **step]**	**[start : end** : **step]**	NA. (**unordered)**
Replication (Multiplication)	str1=ele*multiplier	lst=[ele]*multiplier	tup1=(ele,)*multiplier	NA. Set items are **unique**.
Reverse iterable	str=**str[::-1]**	lst=**lst[::-1]** | lst.**reverse()**	tup=**tup[::-1]**	NA. (**unordered)**
Concatenation (2 or more)	(+) operator. **str1+str2**	(+) operator. **lst1+lst2**	(+) operator. **tup1+tup2**	(**union**) **set1.union(set2)**
Copy (from another)	(=) operator. **str2=str1**	<olst>**.copy()** | **list**(olst)	tup2=**tuple**(tup1)	s2=s1**.copy()** | s2=**set(s1)**
Clear all elements	**str1=""**	lst.**clear()** | lst=**[]**	tup=**tuple()** | tup=**()**	set1.**clear()** | set1=**{}**
Del iterable	**del str1**	**del lst1**	**del tup1**	**del set1**
Add a new element by **name**	(+) operator. **str1+=**ele	ls.**append**(ele) | **ls+=**[ele]	(+) operator. **tup+=**(ele,)	(**add** method) s1.**add**(ele)
Remove element by **name**	NA. Use **replace()/regex**	l1.**remove(e)**	NA. workaround: **via List**	s1.**remove(e) | discard(e)**
Remove element by **index**	**str1[0:i]+str1[i+1:]**	del l[i] | l.pop(i) | l[:i]+l[i+1:]	**tup1[0:i]+tup1[i+1:]**	NA. (**Un-ordered**)
Update element by **index**	**str1[0:i]+'e'+str1[i+1:]**	**lst1[0:i]+[e]+lst1[i+1:]**	**tup1[0:i]+(e,)+tup1[i+1:]**	NA. (**Un-ordered**)
Insert ele. at specific **index**	**str1[0:i]+'e'+str1[i:]**	**lst1[0:i]+[e]+lst1[i:]**	**tup1[0:i]+(e,)+tup1[i:]**	NA. (**Un-ordered**)

B. Specific Methods

1. String (specific)	Methods / Syntax	
Case switching	('**capitalize**', 'Joydeep basu')|('**casefold**', 'joydeep basu')|('**swapcase**', 'jOYDEEP basu')|('**title**', 'Joydeep Basu')|('**upper**', 'JOYDEEP BASU')|('**lower**', 'joydeep basu')|('**istitle**', False)|('**isupper**', False)|('**islower**', False)	
Searching Substring	**First** occurrence: s.**find**(e) | last (reversed) : s.**rfind**(e)	s.**find(e,start,end)** returns **-1** if e not found
Character type determination	**isdecimal**(), **isdigit**(), **isnumeric**(),**isalnum**(),**isalpha**()	**isidentifier**(), **isascii**(), **isprintable**(), **isspace**()
String **split/join** (list/set/tuple)	s.**split(**delimiter**)** >> **list** (default) | **rsplit**() | s.**splitlines**()	<delim>.**join**(iterable) | <ite>.**join**(map(str, ite))
String **replace** (substitute)	s.**replace(**old_value, new_value, count[=*all occurrences]***)**	Note: An alternative of regEx **sub()**
String **Stripping** & **Filling**	Strip: **strip** (both side) | **lstrip** (left) | **rstrip** (right)	Fill: s.**center|ljust|rjust**(width,char), s.**zfill**(char)
String **Translation**	txt.**translate**(**str.maketranslation**(**from, to**, omit))	Here **(from,to)** is a **ascii map** used in translation
Others methods	s.[r]**partition(**delim**)**, s.**startswith**|**endswith(**delim**)**	
2. List (specific)	Methods / Syntax	
All **list supported methods**	**count(), index(), remove(), clear(), append(), extend(), insert(), reverse(), copy(), pop(), sort()**	
Elements **Sorting**	**sort()** – sorts a list. Ex. **lst1.sort()**	lst1=lst1.sort() is WRONG! Only, **lst1.sort()** is OK.
Reversing list	**reverse() –** lst1.**reverse()** or lst1=**lst1[::-1]**	lst1=lst1.reverse() is Wrong! Just, **lst.reverse()**
Remove element (by value or index)	l.**remove**(**val**) (throws error if not found) | l.**remove**(**lst[idx]**) | **del** l[idx] | l.**pop(idx)** | **lst=lst[:idx]**+**lst[idx+1:]**	
Insert a new element	lst**.insert(index,element)**	Alternatively, use slicing: **lst[idx:idx]=[ele]**
3. Tuple (specific)	Methods / Syntax	
All **tuple supported methods**	**index(), count()** [and **slicing** works same as list.]	
Add/Remove/Insert/Delete/.., Alternative of slicing ("**via List**")	Tuples are **Immutable**, but can easily be converted into **lists** which is **Mutable**. So these operations can be performed "**via List**" i.e. **tup=(1,2,3,)** → **lst=list(tup)** → lst.**insert|remove|pop|append**() → **tup = tuple(lst)**	
4. Set (specific)	Methods / Syntax	
Basic **element** operations on set	**add()**→(new item), **remove()**→(specific item), **discard(), pop()**→(random item), **clear(), copy()**	
Combined distinct items of set1&2	**union()** → set1.**union(set2)** or set1 **|** setB	**update()** – updates **union** on set1
Common items of both set1&2	**intersection()** → set1.**intersection**(set2) or set1 **&** set2	**intersection_update()** – updates **inter..** on set1
Combined Unique items of set1&2	**symmetric_difference()** → set1.**s..d..**(set2) or set1**^**set2	**symmetric_difference_update()** – updates (←)
Unique items of set1 NOT in set2	**difference()** → set.**difference**(set2) or set1 **-** set2	**difference_update() –** updates **diff..** on set1
Check whether **Set1.method(Set2)**	**isdisjoint() -** (no intersection?)**, issubset()** (s1 if s.s of s2)	**issuperset()** whether set1 having all set2 items
Create a **Frozenset** (immutable set)	**frozenset()** → **fset1 = frozenset(iterable)**	Note: frozenset has no **add/remove** methods
5. Dictionary (specific)	Methods / Syntax	
Create new Dict. from list/tup (keys)	d1=**dict.fromkeys(**iterable, default value**) | Iterable:** tuple/list of **keys** | Value: can be anything, **0** | **[]** | **None**	
Copy from a dictionary	**d2=d1.copy()** or **d2=dict(d1)**	**d2=d1** with just create a name ref. & not a copy.
Get **key/value/items** list	**keys()** → **list** of all keys **| values()** → **list** of values	**items()** → returns a **list** of (key, value) tuple
Set/Get/Update value by key	**d1[key] = val** | **d1.get(key)** | d1.**update({key:val})**	**clear()** → clear all **keys & values** of a dict (reset)
Remove/Pop element	x=d1.**pop(key**, *[return value if key missing]*)	**popitem()** removes the **last key** & returns **(k,v)**
Return **default** value if key missing	d1={1:10,2:20}; d1.**setdefault(2,25)** will return **20** but **d1.setdefault(3,35)** will return **35** as key **3** is missing.	

C. Built-in Functions

Iterator Functions	Description	Example/Note
sorted()	Return a **new sorted list** from the items in iterable.	**[x for x in sorted(tup, key = lambda x: x[idx])]**
enumerate()	Takes a **collection**-adds a **counter** (key)-return **enumerate object**	**d1={k:v for k,v in enumerate(**str/tup/lst, ***start*=0)}**
filter()	Removes un-matched elements from iterable. **list(filter(lambda x: x>=10 and x<=80, [5,10,20,70,90]))** >> **[10, 20, 70]**	
reversed()	Return a reverse iterator object. Use list()/tuple() to extract items	**list(reversed([1,2,3,4,5]))** >> **[5,4,3,2,1]**
map()	Return an iterator that applies function to every item of iterable.	Syntax: **map(function to apply on each, iterable)**
lambda()	**Single line anonymous function** defined without any name.	res = list(map(**lambda x: pow(x,2)**, iterable))
all()	Return True if **all elements of iterable** are **true**. *l3= [10,20,30];* ***all****(map(lambda x: True if x%3==0 else False,l3))* >> ***False***	
any()	Return True if **any element of iterable** is **true**. *l3= [10,20,30];* ***any****(map(lambda x: True if x%3==0 else False,l3))* >> ***True***	
zip()	Iterate over several iterables in parallel.	**res={k:v for k,v in zip(lst1,lst2)}** *(len(lst1)=len(lst2))*

iter() | aiter()
next()
	Return an **iterator** object. [**aiter()** for **asynchronous** object] | **next()** Retrieve the **next item** from the iterator. For example, x = **iter(["Joy", "Deep", "Basu"])** | **type(x)** >> <class '**list_iterator**'> | **next(x)** > Joy; **next(x)** > Deep; **next(x)** > Basu	

3. Topics:

A. Comprehensions

1	If-else	print(True if len(x) >= 7 else False) | print(True) if len(x)>=7 else print(False)	if condition can be nested.
2	List	l1 = [x for x in iterable if x%2==0] l2= [odd.append(x) if x%2!=0 else even.append(x) for x in iterable if x<100]	Condition can be placed in both with expression & iterator loop
3	Dictionary	odict = {x:x**3 for x in input if x % 2 == 0}	{k:v for (k, v) in iterable if (cond)}
4	Set	oSet = {x for x in input if x % 2 == 0}	
5	Generator	print(x for x in input if x % 2 == 0) >> Prints a <generator object>, use type conversion list()/tup()/set() on this.	

B. Conditional Statements

1	if…elif…else	General if <condition1>: \n <code> elif <cond2>: \n <code> elif <cond3>: \n <code> \n else: \n <code>
2	match…case	Python v3.10 supports match <exp> \n case <val>: <code> \n case <val>: <code> \n case default: <code>

C. Functions & Arguments

1	Function as "Object"	Any custom name (here, cf) can be assigned on a lambda function to use that function as object. Ex: cf=lambda x,y,z: x+y-z | print(cf(10,20,5)) >> 20 | Note: lambda() doesn't require a return keyword.	
2	Function as "argument"	def add(x, y): return x + y | def sub(x, y): return x - y | def ope(func): return func(100,20) list(map(ope, [add, sub])) returns [120, 80] | Here, function add and sub are passed to ope as "argument".	

3	Function as "return value"	def master(ope):
		def add(x, y): print(x + y)
		def sub(x, y): print(x - y)
		return eval(ope)
			x = master('add') # here, x is the 'add' function returned by master
			y = master('sub') # here, y is the 'sub' function returned by master
			Now, calling → x(5,9) will print 14
			And calling → y(10,5) will print 5

4	*args vs **kwargs and Optional/default argument	To pass unknown/arbitrary number of arguments (*args) or keyword arguments (**kwargs) in a function. Default arguments are passed with a value, so, if not provided during function call it will use the default value.	

D. Loops

1	for x in range(start,end,step)	for x in range(0,0) → doesn't loop | for x in range(0,1) → loops for x=0 | for x in range(len(iterable)) for x in range(0,10,2) → loops for x=0,2,4,6,8 (as increment step is s) | Note: loop runs for x=start,,,end-1
2	for x in iterable	tuple(0,0): for x in (0,0) | list: for x in [0,1,0] | dict: for x in d.keys()|values() | for x in enumerate(txt) | etc.
3	for else loop	If for loop faces a break statement it does NOT execute the else part. Otherwise, else part is executed after all for loop iterations.
4	while loop	while condition → while True executes the loop but while None doesn't execute the loop lst=[]; while lst: print(True) → this loop will not run because lst is empty >> while None:
5	break vs continue	break → exits the loop immediately when meets breaking condition. continue → skips the remaining loop statements and goes to the next iteration of the loop.

E. Class-Object-OOPs

1	Class & Object	A Class is a like a blueprint/prototype for an object and an Object is an instance of a class.		
2	Instance vs class components methods-attributes-keyword	1. Keyword	cls for class	self for instance/object
		2. Decorator	@classmethod	NA
		3. Access class v instance methods & attributes	cls./self. (from inside or within class) className. <method/attr> (outside)	self.<method/attr> (from inside) objName.<method/attr> (outside)
3	Constructor (__init__)	Calling a class first invokes the constructor __init__() method. super().__init__() is parent class constructor.		
4	Access Modifiers on method/attributes	Three – 1.Public|2.Protected|3.Private → starts with 1.No|2.Single|3.Double underscore and accessible from 1. anywhere of the program | 2. Class & inherited child classes |3. the defining class only.		
5	OOP > Inheritance	Super/Parent/Base class being inherited by Sub/Child/Derived class to inherit parent methods & attributes		
6	OOP > Multiple Inheritance	Python allows Multiple Inheritance by a child class but it follows Method Resolution Order (MRO).		
7	OOP > Encapsulation	Class as a grouping or 'Wrapper' of all variables & Methods declared inside it.		
8	OOP > Polymorphism	1. Same Base class methods being overriden differently by inherited subclasses. 2. Same class method results differently for different data types. Ex. + operator → Addition-Concatenation		
9	OOP > Abstraction	1. from abc import ABC, abstractmethod 2. use @abstractmethod decorator above base/parent class method and pass in the body | 3. override the same method from child class.		

F. Closures & Decorators

1	Decorator	To execute pre-requisite methods/events on before/after the original function decorators are used.
2	Closure	Closure is like function as Object, an inner function defined inside of outer function.
3	Decorator implementation	1. Function as decorator, 2. Class as decorator (see example later)

G. Error Handling

1	Exception Raising	1. raise Exception|ErrorType(error_msg) | 2. assert(condition) → error is thrown if condition is not TRUE.
2	Exception Handling	1. Generic: try – except 2. try – except (specific error type) 3. try – except – else – finally
3	try – except – else – finally	2 possible workflows based on condition met or failed in try block. 1: try-except-finally 2. try-else-finally

H. Threads & Process

1	Features	Multi-Threading	Multi-Processing
2	Import Library	from threading import Thread	from multiprocessing import Process

3	1. Define (Target function)	def targetFunction(arg): <some code>
		lst=[] | for x in range(N): # N or use data iterable
		lst.append(Thread(target=doSomething,args=(x+1,)))
			def targetFunction(arg): <some code>
			lst=[] | for x in range(N): # N or use data iterable
			lst.append(Process(target=doSomething,args=(x+1,)))

4	2. Start	for idx in range(len(lst)): lst[idx].start()	for idx in range(len(lst)): lst[idx].start()
5	3. Join	for idx in range(len(lst)): lst[idx].join()	for idx in range(len(lst)): lst[idx].join()
6	PoolExecutor (concurrent.futures)	with concurrent.futures.ThreadPoolExecutor() as tex: tex.map(doSomething,[x for x in data_iterable])	with concurrent.futures.ProcessPoolExecutor() as pex: pex.map(doSomething,[x for x in data_iterable])

4. Special Utilities/Libraries

A. Collections

1	**Counter** (Dictionary)	It is a **Built-in Frequency Counter** (from Collections library), a **dictionary** that stores **element:frequency** as **key:value** when assigned any string/list/tuple. For ex. **Counter([1,2,3,2,-1,-2,-3])** returns **Counter({2: 2, 1: 1, 3: 1, -1: 1, -2: 1, -3: 1})**. A dictionary that can store elements as key & its appearance count as value.
2	**DefaultDict**	It is a **dictionary** declared with (default type: **list |set | dict |tuple**), so that any non-existing key with return an **empty datatype** as default value. Ex. d=**defaultdict(list);** d[12] >> **[]**
3	**Deque**	A **double-ended queue** used to add or remove elements from **both ends**. Deque methods are – 1. **insert**(index, element) 2.**append[left]** (right/left), 3.**pop[left]** (right/left), 4. **extend[left]** (right/left), 5.**reverse**, 5.**rotate** (**x**) – **Right to Left** rotation.
4	**NamedTuple**	Access tuple elements using **attribute name** instead of index. List of attributes are defined in sequence for the **named** tuple. **Student = namedtuple('Student', ['name', 'age', 'DOB']); nandu = Student('Nandini', '19', '2541997')** **print(f"{type(nandu)} | {nandu.name}")** >> type(nandu): <class '__main__.Student'> | Nandini

B. Itertools

1	**permutations**	1. **N length** permutation on iterable (arr) → **list(permutations(arr,length=N))** 2. All possible permutations (of any length) on iterable (arr) → this does not require length argument. **allPerms = [] (\n) for i in range(1, len(arr)+1): allPerms.extend([x for x in permutations (arr,i)])**
2	**combinations** and **combinations with replacement**	1. **N** length combinations on iterable (arr) → **list(combinations(arr,length=N))** 2. **Powerset**: All possible combinations (of any length) on iterable (**arr**) → this does not require length argument. allCombs = [] (\n) for i in range(1, len(arr)+1): **allCombs.extend(list(combinations(arr,i)))** 3. **N** length **combination with replacement** → **list(combinations_with_replacement (arr,length=N))**
3	**groupby**	It is **consecutive occurrence counter** that returns **(frequence, element)** tuple for the iterable being grouped by. For Example. num = "1222313333442"; print(**[(len(list(g)),int(k)) for k,g in groupby(num)]**) >> [(1, 1), (**3, 2**), (1, 3), (1, 1), (**4, 3**), (**2, 4**), (1, 2)]
4	**product**	**product()** returns a "**sorted list**" - the **cartesian product** of input iterables. **list(product([1,2,3],[10,20,30],[100]))** >> *[(1, 10, 100), (1, 20, 100), (1, 30, 100), (2, 10, 100), (2, 20, 100), (2, 30, 100), (3, 10, 100), (3, 20, 100), (3, 30, 100)]*

C. functools, operator

1	**functools.reduce()**	**lst=[1,2,30,4,55,6,7,8,9]; print(reduce(lambda x,y: x+y, lst)) >> 122 # calculates sum of the iterable** **Max: reduce(lambda x,y: x if x>y else y, lst) >> 55 |Min: reduce(lambda x,y: x if x<y else y, lst) >> 1**
2	**reduce** with **operator**	lst = [1,2,30,4,55,6,7,8,9]; print(**reduce**(**operator.add**, **lst**)) >> **122**

D. Numpy

1	Import **Library**	import **numpy** as **np**
2	**Numpy array** (ndarray)	**nd** = **numpy.array(iterable, dtype***[=float]*) → same for **1|2|3|N-dimension** | **nd.ndim** returns **dimension** of **nd** *Note: default dtype or datatype value is **float** (So, if **dtype** argument not provided it will create float nd array)*
3	Accessing **elements**	For, 1D → **nd**[index]; 2D → **nd**[*row_index, column_index*]; 3D → **nd**[*array_index, row_index, column_idx*]
4	**Modify** dimension	1. **nd.shape**=(rows,cols) | 2. **nd.reshape**(rows,cols) | 3. **numpy.transpose(nd)** | 4. **nd.flatten()** Note: methods **transpose, reshape, flatten** does NOT modify **original array** but **shape** does.
5	Create **Predefined dimension** methods	1. nd=numpy.**zeros**((rows,cols), dtype) 2. nd=numpy.**ones**((rows,cols), dtype) → creats ndarray with **0s** | **1s** 3. nd=numpy.**identity**((**N**), dtype) → returns a **NxN square matrix** 4. nd=numpy.**eye**((**rows, cols, k**), dtype) → returns **2D matrix** with k=diagonal position **1s** and rest all as **0s**
6	**Concatenate** on **axis**	syntax: numpy.**concatenate**((**nd1,nd2**), **axis**=**0|1|n**) → performs addition of 2 ndarray elements as per axis given.
7	**Math & Statistical** operations	A. **nd/2** or np.**divide**(nd, 2) | B. **nd1** (+ - * / ** %) **nd2** → np.**add|substract|multiply|divide|power|mod**(nd1,nd2) C. np.**floor|ceil|rint**(nd) | D. np.**sum|prod|min|max|mean|var|std**(nd, **axis**=**0|1|None**)
8	**Vector** operations	**l1, l2 = [1,2] , [3,4]** ; **A** = **np.array(l1)** ; **B** = **np.array(l2)** 1. **Dot** product: **np.dot(A,B)** | 2. **Cross** product: np.**cross**(A,B) | 3. **Inner** → np.**inner**(A,B) | 4.**Outer** → np.**outer**(A,B)

E. Regular Expression (re)

1	**Regex Methods**	1. re.**search**(exp,txt) returns a **re.Match** object with method/props like, **start(), end(), span(), string, group|s()** 2a. re.**match**(exp,txt) returns match object if pattern found **anywhere in the input text**. 2b. re.**fullmatch**(exp,txt) returns match object only if the text fully matches the pattern else return **None**. 3. re.**findall**(exp,txt) → **list** of all matching result. 4. re.**sub**(old chars**,** new chars**,** txt**,** occurrence) – replaces old text with new text as per occurrence (default = all). 5. re.**split**(delimiter, txt, max_occurrence) → returns a list of strings after the spilt by specified delimiter.
2	**Meta Characters**	1. **[]** →Set of chars **[acf]**, 2. \ →Escape special chars **\d \s**, 3. **|** →either or **s|d**, 4. **^**→ Starts with, 5. **$** →Ends with 6. **()** →Capture group, 7 **.** →any char, 8|9|10. *****(0 or more) **?**(0 or 1) **+**(1 or more) occurrences, 11. **{min,max}** → exact
3	**Special sequences**	These returns a **match** where found. **\d** (numbers 0-9) **\s** (space) **\w** (a-z,0-9,_) **\A** or **\Z** (chars starts or ends with) Returns **match** for the **opposite** cases. **\D** (not digits) **\S** (not space) **\W** (not a-z0-9_) **\B** (chars not at start or end)
4	**Sets**	1. Specified chars **[arn]** or **[012]** 2. In Range **[a-z]** or **[0-9]** 3. Not in [^exp] 4. **[0-5][0-9]** number between 00 to 59
5	Regex **group()**	It returns one or more subgroups of the match.
6	Look(**ahead|behind**)	A. Positive **lookahead**: **(?=<lookahead_regex>)** B. Positive **lookbehind**: **(?<=<lookbehind_regex>)**
7	Regex **flags** argument	**flags** argument is used to modify search behaviour: flags = re.**IGNORECASE** (Case Insensitivity) | re.**DOTALL** (Dot Matching Newline \n) | re.**MULTILINE** (**Multiline Mode**) | Verbose Mode| Debug Mode

F. HTML/XML Parsers

1	**html parsing using** HTMLParser **library**	1. Import lib: from **html.parser** import **HTMLParser ;** 2. Create a custom class**:** myhtmlParser(HTMLParser)**,** 2. Override methods: **handle_(**starttag**|**endtag**|**startendtag**|**comment**|**data**|..)** using arguments**:** tag, attrs, data 3. Create parser object: chp = **myHtmlParser();** 4. Feed html-string**:** chp.**feed**(html_str)**;** 5. Close parser: chp.**close()**
2	**XML parsing** using xml2dict&ElementTree	Lib 1. import **xml2dict**; print(**xml2dict.parse**(**xml_string**) Lib 2. import **xml.etree.ElementTree** as **etree ;** tree = etree.**ElementTree**(etree.**fromstring**(**xml_string**)) ; root=tree.**getroot**() ; print(**root.attrib**) → Now, loop recursively for child nodes ie. for **child** in **root**: print(child.**attrib**)

5. Key Notes

SL	Category	Note
1	General	print(dir(**builtins**)) # prints a list of all python reserved built-in variables/names
2	Variables	Variable scope: The "**LEGB**" rule --> Local, Enclosing, Global, Built-in (order of overriding).
3	General	We can modify the **default delimiter (\n)** between 2 **print** statements using **end** parameter.
4	function (built-in)	**exit()** terminates the **whole program execution** irrespective of whether called in main() or any sub functions().
5	map + datatype funcs	To use data type specific functions with map, use **datatype.function** name inside **map**. Ex. map(**str.upper**,iterator)
6	global statement	use **global keyword** before variable name inside the local function to refer to the global instance of the variable.
7	break vs continue	break **exits the current loop** and continue **skips remaining steps** of current iteration & proceeds to next iteration.
8	switch case	Introduced in python **3.10 onwards** but not supported in earlier versions.
9	exit() or sys.exit()	Stops the entire execution wherever called - using **sys.exit()** or built-in function **exit()**.
10	String literals (**f** | **r** | **u**)	**f-literal** prints embedded **{expression}** along with string. **r-literal** prints raw string (including backslash) used for regular expression. **u-literal** is used for Unicode chars.
11	String split() delimiters	**split()** by default splits substrings **separated by one or 'more' spaces**. **split("")** throws empty delimiter exception.
12	String element search	If substring or element is missing, <str>.**find**(e) returns **-1** but <str>.**index**(e) throws **exception**, so **avoid index()**.
13	String partition vs split	string [r]**partition(delimiter)** always splits input into **3 parts** – 1. Before part, 2. delimiter, 3. After part, even if delimiter found in after part – it ignores which is not the case for split().
14	Slicing Rule	**1. Forward Index** = **Reverse Index** + **length** of Iterable. 2. Slicing happens **before End** index. Ex. lst**[0:3]**>**0,1,2**
15	Single element Tuple	**tup1 = (1,) Must use comma** if only **one value** is present else will be considered as other data type.
16	Message Formatting	1. Using **f-string** print(f"msg{exp}"), 2. Message with {placeholders}.**format**(named **argument | tuple | Dictionary**)
17	Round vs Format	For decimal place formatting - **round()** does NOT give correct results always, but **format()** does.
18	List index exception	If element not present **lst.index(ele)** returns exception, to avoid error use prefix: **if ele in lst** first
19	List copy from another	lst2=lst1 **copies the name only** without creating a new list, so any change in **lst2** with update the original list (**lst1**) To create an **independent** copy without affecting the original, use either A. lst2 = **lst1.copy()** or B. lst2 = **list(lst1)**
20	Iterable–create a **copy**	Use type conversion function on the original one i.e. **list(), tuple(), set(), dict()** creates a **new** copy of the original.
21	Use of **Asterix (*)**	1. Elements **unpacking** (***iterable**), 2. To pass **arbitrary/unknown** no of arguments (***args**), 3. To pass arbitrary/unknown number of **keyword arguments** (****kwargs**), 4. Calculate **Multiply** (*), 5. Calculate **Power** (**)
22	Multi-variable assignment at once	Use tuple to assign multiple variables at once. Ex. **var1, var2, var3 = tup** (here, **tup** must contains **3 values** only) If **tup** contains more than 3 values, write var1, var2, ***var3** = tup (to assign all rest from 3rd onwards as **list** for var3)
23	Iterable conversion & operation together	**tup=(1,2,3,); tup=list(tup).insert(0,0)** → will throw error but **l=list(tup); l.insert(0,0); tup=tuple(l)** will work fine. Iterable conversion & operation on it **at the same time fails**. Hence, perform them in separate lines.
24	Set operation representation and important notes	Set operations exactly follows to **Venn-diagram** of two or more circles. 1. **Union/Intersection/Symmetric difference** are **NON-Directional** i.e. returns **same result** for A on B or B on A. 2. Difference method is **directional** i.e. **(A-B)** and **(B-A)** returns **different results**. 3. **Symmetric difference** = **Union – Intersection**. 4. Set operations for **more than 2 sets** are possible. **set1**.union(**set2,set3,set4); set1**.intersection**(set2,set3,set4)** 5. Two sets are **disjoint** when they have **no common elements** so their intersection returns null.
25	Set **remove()** vs **discard()** vs **pop()**	Set's **pop()** removes any **random element** from set, it takes no argument. But remove/discard uses specific name. **remove(ele)** method throws **exception** if **element is not found** within the set but **discard(ele)** returns **None**.

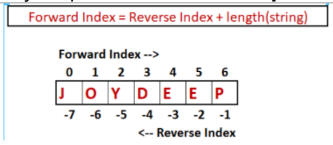
26	Sort Dictionary by Key and Value	Sorted by **Key**: **{k:v for k,v in sorted(d1.items(),key=lambda x: x[0])}**
		Sorted by **value**: **{k:v for k,v in sorted(d1.items(),key=lambda x: x[1])}**

27	**Counter** use cases	Use Counters when calculations are related to **element frequency count** like – alphanumeric | word | line count.
28	**Default Dictionary** use cases	Use **defaultdictionary** in place of normal dictionary when one dictionary key may contain multiple **values** that needs to be stored in some iterable/list. Useful for **nested iterables**.
29	**Deque** uses case	**Array rotation** cases or both end element operations.
30	**hash()** note	**hash()** works in **immutable** objects and **returns exception** when used on **mutable** objects. A. **Immutable**: **hash((1, 2, 3,-4))** | **hash("Joydeep")** → valid. B. **Mutable**: **hash([1,2,3,4,5,6,7,8])** throws exception
31	**map()** vs **reduce()**	Both **map & reduce** works on iterable **elements** but reduce **reduces the iterator** to a **single return value** by comparing among elements **consecutively** while map runs for **every element** & returns an **iterator object**.
32	**Class** notes	1. **Class Namespace** - **ClassName.__dict__** will show all class attributes/methods like a **dict key:pair** 2. **Class attributes** are COMMON to all i.e. object/instance independent. 3. **Instance attributes** are initiated inside class Constructor (__init__) and requires self keyword to access them. 4. **self keyword** is **mandatory** in all instance methods of a class as the first argument. 5. **super().__init__()** - Used within **child class constructor to instantiate parent class constructor** with matching arguments provided in the child class constructor. It does NOT require any 'self' keyword as first argument. 6. **isinstance(object, class)** - return **True/False**, checks if the variable/object is an **instance** of the specific **class** or not. If object created from child class, it returns true for both **super class** as well as **child class**. 7. **__repr__()** defines the **string representation** of a class. There are some **dunder** methods.
33	**Oops** notes	1. Python does not support **method overloading** what java does.
34	Decorator **Chaining**	When multiple decorators are used on a function it is called decorator chaining.
35	**Closure** notes	It is a **nested function**. The closure will have **access to a 'free' variable** that is in outer scope. It will be returned from the **enclosing (outer) function**. Closure can be called a **function object** (function as **object**) that is capable of **remembering values** that are in **enclosing scopes** (outer functions) even if they are not present in memory.
36	Multi-**Thread|Process**	1. **Multithreading** are good for **I/O bound tasks** where as **Multiprocessing** are good for **CPU-bound tasks**. 2. **Multi-threading** adds **some overhead** but **multi-processing** does NOT and therefore **truly concurrent**.
37	Thread/Process **join()**	join() **pauses code execution** until **completion** of thread/process else it will **proceed** to **end** or **next code block**.
38	**Pool Executor**	concurrent.futures.(Thread|Process)**PoolExecutor**() → **Context manager** that automatically handles **joining**.
39	**Date/Time** libraries **Timezone** operation	import **datetime** as **dt** | from **dateutil** import **tz** **fmt**='%a %d %b %Y %H:%M:%S %z'; **dt1** or **dt2** = **dt.datetime**.**strptime**(date1|date2,fmt).**astimezone**(**tz.tzlocal()**) **diff** = **dt1 - dt2** if **dt1 > dt2** else **dt2 - dt1** ; to calculate difference in seconds → **diff.total_seconds()**
40	Set element **update**	set1**.discard(old_element)**; set1**.add(new_element)**

6. General Examples

#	Topic / Context	Example

1	**Scope** of variables - **LEGB rule**	x = 'global x'
		def outer():
		    # **global x** # if enabled, this will update the global variable (x) value from 'global x' --> to 'local x'
		    x = 'local x'
		    def inner():
		        # **nonlocal x** # if enabled, this will update the local variable (x) value from 'local x' -> to 'enclosing x'
		        x = 'enclosing x'
		        print(x)
		    inner() # prints the 'enclosing x' (enclosing scope) when nonlocal x is NOT used
		    print(x) # prints the 'global x' (global scope) when global x is NOT used inside outer function

2	String **stripping** unwanted characters	' Joydeep Basu '**.strip()** > **Joydeep Basu** | '$$$Joydeep$$$Basu###'**.strip('$')** >> **Joydeep$$$Basu###** '$$$Joy$$$Basu##'**.lstrip('$')** > **Joy$$$Basu##** | '$$Joy$$$Basu###'**.rstrip('#')** >> **$$Joy$$$Basu** '$$$Joydeep$$$Basu########'**.lstrip('$').rstrip('#').replace('$$$',' ')** >> **Joydeep Basu**
3	String **filling** (justifying text)	Joydeep Basu **.center(50,'-')** >> **-------------------Joydeep Basu-------------------** Joydeep Basu **.ljust(50,'-')** >> **Joydeep Basu--------------------------------------** Joydeep Basu **.rjust(50,'-')** >> **--------------------------------------Joydeep Basu** Joydeep Basu **.zfill(50)** >> **00000000000000000000000000000000000000Joydeep Basu**
4	String **Replacing** characters	#####Joydeep**Basu###### **.replace('#','')** >> **Joydeep**Basu** #####Joydeep**Basu###### **.replace('*','',1)** >> **#####Joydeep*Basu######** #####Joydeep**Basu###### **.replace('#','').replace('*','',1).replace('*',' ')** >> **Joydeep Basu**
5	String **Expand Tabs** function	J O Y D E E P **.expandtabs(0)** >> **JOYDEEP** J O Y D E E P **.expandtabs(2)** >> **J O Y D E E P** J O Y D E E P **.expandtabs(4)** >> **J O Y D E E P**
6	String **Translate** & **Make translation** function (encoding)	**str.maketrans('e','i')** > (map1) > {101: 105} | Joydeep Basu **.translate(map1)** **Joydiip Basu** str.**maketrans**('JB','jb') > (map2) > {74: 106, 66: 98} | Joydeep Basu **.translate(map2)** **joydeep basu** str.**maketrans**('JBe ','jbi*', **'ou'**) > (map3) > {74: 106, 66: 98, 101: 105, 32: 42, **111: None**, **117: None**} Joydeep Basu **.translate(map3) >>** **jydiip*bas**
7	Iterable **Slicing** rules	 Forward Index = Reverse Index + length(string) Forward Index --> 0 1 2 3 4 5 6 J O Y D E E P -7 -6 -5 -4 -3 -2 -1 <-- Reverse Index name**[0]** or name**[-7]** >> **J** |name**[6]** or name**[-1]** >> **p** name**[:]** or name**[0:7]** = **Joydeep** | name**[-7:0]** >> (null) name**[:3]** or name**[:-4]** = **Joy** | name**[3:]** or name**[-4:]** >> **deep** name**[3:5]** or name**[-4:-2]** = **de** (End index element not counted) name**[::-1]**=**peedyoJ** (reversed) |name**[::2]** = **Jyep** | name**[1::2]** = **ode**

8	Print message formatting using **tuple, dictionary** & **named arguments** with **format()**	**valTuple1** = ('Joydeep','39'); **valDict1** = {'name':'Jui','age':18}
		print('Hi, I am **{}**, and my age is a variable, as of now, **{}** years.\n'
		'And **{name}** is my wife and her age is a constant, always **{age}** years.'**.format(*valTuple1, **valDict1)**)
		print('My name **{name}** and age is **{age}** years.'**.format(name='Joydeep',age=39)**)

9	Logical operators (**and, or, not**) within if (**in range, is None**)	**x:10, y:3** | x if x>y else y >> **10** True if **(x>=10 and y<=5)** else False >> **True** | True if **(x>=10 or y<=5)** else False >> **True** True if not **(x>=10 and y<=5)** else False >> **False** | True if **not (x is None or y is None)** else False >> **True** True if (**x in [5,10,15] and y not in [1,2,3]**) else False >> **False** True if (**x in range(11) and y not in range(3)**) else False >> **True**
10	Element Unpacking (***iterables**)	if lst=[1,2,3,4,5,6,7,8] then, **' '.join(map(str,lst))** is same as print(***lst**) → space separated string elements.

11	Function (***args**)	def func1**(*args):**
		        return sum(args) | print(f"**Unknown no. of arguments** sum: {**func1(10,20,30,40)**}") >> **100**

12	Function (****kwargs**)	def func2(****kwargs**): for key in **kwargs**: print(f"Value of [{**key**}] is:{**kwargs[key]**}") func2(**fname="Joy"**,**lname="Basu"**,**age=39**) >> Value of [**fname**] is:**Joy** Value of [**lname**] is:**Basu** Value of [**age**] is:**39**
13	Generate 2D numpy matrix (0s)	numpy.matrix(**[[0 for x in range(10)] for y in range(10)]** or (**[[0]*10]*10**)) >> numpy matrix (list of lists).
14	Generate a Panda Dataframe	pd.**DataFrame**(**[{'param1':'val1','param2':'val2','param3':'val3'} for y in range(10)]**) >> (list of dicts)
15	Dictionary of lists Nested Dictionary	1. **Dictionary of Lists**: {y:[] for y in range(10)}; 2. **Nested dictionary**: nestedDict = {y: {'param1': 'val1', 'param2': 'val2', 'param3': 'val3'} for y in range(10)}
16	List **extend()** vs **append()**	d1, rExt, rApp = {1: [10,20], 2: [4,11], 3:[0,20]}, [], [] for x in d1.values():**rExt.extend(x)** >> [10, 20, 4, 11, 0, 20] → extend **adds the elements** of a new list. for x in d1.values():**rApp.append(x)** >> [[10, 20], [4, 11], [0, 20]] → append **adds the new list** itself.
17	String/Tuple/List/Set to Dictionary using **enumerate()**	txt = 'Joy deep Basu'; lst = ['Joy', 'Deep', 'Basu']; tup = ('Joy', 'Deep', 'Basu'); set1 = {'A','E','I','O','U'} **print({k:v for k,v in enumerate(txt|lst|tup|set1, 0)})** # **string|list|tuple|set** to **dictionary**
18	**For Else** loop	For **N=3** → the **else print** statement will NOT be executed (**since loop is breaking**) but for **N=7** it will. **for x in range(5):** if x>=**N**: **break;** print(f"Current value: {x} is less than break condition {**N**}") **else:** print(f"Current loop did not reach break condition, so else block is executed")

19	**break** vs **continue**	i, lst = 0, []
		while **i <=10**:
		  if **i == 5: break** # Keyword break **exits the loop at 5**, so 6,7,8,9,10 not executed.
		  i+=1
		  if **i == 3: continue** # this skips next 'append' step & proceed to next iteration → **3** not appended to **lst**
		  lst.append(i)
		print(lst) >> **[1, 2, 4, 5]**

20	**match..case** (supported in python version **3.10** onwards)	**match exp**: **case 0**: return "zero" \n **case 1**: return "one" \n **case 2**: return "two" \n **case default**: return "something"	
21	**reversed()**	txt = 'Joydeep'; tup = (1,2,3); lst = [1,2,3]; list(reversed(txt)) → ['p', 'e', 'e', 'd', 'y', 'o', 'J'] | list(reversed(lst)) → **[3, 2, 1]** | print(list(reversed(tup)))>**[3, 2, 1]**	
22	Use of '**end**' in print statement default value of **end** is **'\n'**)	lst1 = [1,2,3,4,5,6,7,8,9] >> for x in lst1: print(x, **end**='') lst2 = ['My','age','is','40'] >> for x in lst2: print(x, **end**='$')	**123456789** **My$age$is$40$**
23	Print() > **sep** param	**print(1,2,3,4,sep='-')** >> **1-2-3-4**	default value of sep parameter is ''
24	Input() - **default message**	ans = **input('What is your name? ')**; print(ans)	input() is prompting a **default msg.** first before printing user provided input.
25	debugging using **pdb**	**import pdb** | a = 19; b = 0 | **pdb.set_trace()** | addition = a+b | substraction = a-b | **pdb.set_trace() | …** Common PDB commands, # 1. **l** (list, shows all lines and show the debugging cursor (line number) where execution is paused) # 2. **n** (next, executes current line, and moves to next line) # 3. **p** (print, print variable values to check manually) # 4. **c** (continue, executes all the remaining commands until end or a new pdb.set_trace() line found.)	
26	**Function** as **Decorator**	def **decorator_func**(**func**): print("prerequisites"); **return func** # **decorator function** must **return** the original function **@decorator_func** # Here, @decorator function name is passed def **orginal_function**(): print("main") **orginal_function()** >> **prerequisites \n main** # original function **first executes decorator function**	
27	**Class** as **Decorator**	**class deco**: # class decorator that prints the called function result **def __init__(self,func)**: **self.func = func** **def __call__(self, *args, **kwargs):** # wrapper function that executes the orginal function **print(f"class decorator code: {self.func.__name__,*args}")** # Extra code on top of original function **print(f"Output: {self.func(*args, **kwargs)}")** # executes the original function **@deco** **def add(x,y): return x+y** >> **class decorator code before executing: ('add', 10, 20) \n Output: 30**	
28	**Decorator** with **Arguments**	**def p(func)**: # decorator function that prints the called function result **def wrapper**(***args, **kwargs**): # wrapper function that executes the orginal function **print(f"wrapper code to modify:{func.__name__, *args}")** # Extra code on top of original function **print(f"Output: {func(*args, **kwargs)}")** # executes the original function **return wrapper** # returns the original function after decoration/wrapper codes **@p** **def add(x,y): return x+y** >> **wrapper code to modify:('add', 10, 20) \n Output: 30**	
29	regex **search()** vs **match()** vs **fullmatch()**	**search()** & **match()** return match for **partial string match** but **fullmatch()** requires **whole string match**. re.**search("^x", 'x123sd432df'))** returns **<re.Match object; span=(0, 1), match='x'>** re.**match**(**"^x", 'x123sd432df'**)) returns **<re.Match object; span=(0, 1), match='x'>** re.**fullmatch**(**"^x", 'x123sd432df'**) returns **None** but, re.**fullmatch**("**^x\w***",**'x123sd432df'**) returns <re.Match object; span=(0, 11), match='x123sd432df'>	
30	re **split()** subtext extraction	txt='**x123sd432df**' ; exp="**[0-9]+**" ; print(re.**split**(exp,txt)) >> **['x', 'sd', 'df']** #extract text between numbers	
31	re **findall()** numbers extraction	txt='**x123sd432df**' ; exp="**[0-9]+**" ; print(re.**findall**(exp,txt)) >> **['123', '432']** #extract numbers in whole text	

32	re **sub()** replace
	& extract text or numbers
		txt='**x123sd432df**' ; old="**[0-9]+**" ; new="" print(re.**sub**(old, new, txt)) >> **xsddf**
		txt='**x123sd432df**' ; old="**[a-z]+**" ; new="" print(re.**sub**(old, new, txt)) >> **123432**	

33	**Extraction** of digits-letters alphanumerals & special chars from given text	txt=' -_!||{}^:;/\<>*PO Number generated 13982020 successfully****()^!@#$%%_&' 1. **Numbers**: re.**sub**("**[^0-9]**",'', txt) >> **13982020** 2. **Letters**: re.**sub**("**[^a-zA-Z]**",'', txt) >> **PONumbergeneratedsuccessfully** 3. **Spl chars**: re.**sub**("**[a-zA-Z0-9]**",'', txt) >> **`-_!\|\|{}^:;/\<>* ****()^!@#$%%_&** 4. **AlphaNumerals**: re.**sub**("**[^a-zA-Z0-9]**",'', txt) >> **PONumbergenerated13982020successfully**	
34	re match **groups()**	**txt** = '**username@hackerrank.com**' ; **exp**= r'**(\w+)@(\w+)\.(\w+)**' ; **m**=re.**match**(exp,txt); print(**m.groups()**) #>> **('username', 'hackerrank', 'com)** print(**m.group(0)**) #>> **'username@hackerrank.com'** (The entire match) print(**m.group(1)**) #>> **'username'** (The first parenthesized subgroup) print(**m.group(2)**) #>> **'hackerrank'** (The second parenthesized subgroup) print(**m.group(3)**) #>> **'com'** >> (The third parenthesized subgroup) print(re.**findall**(exp, txt)) #>> **[('username', 'hackerrank', 'com')]** (a **list of group tuple**)	

35	re match **groupdict()** with named parameter	**re.match(r'(?P<user>\w+)@(?P<domain>\w+)\.(?P<ext>\w+)','jbasu@hackerrank.com').groupdict()**
		>> {'user': 'jbasu', 'domain': 'hackerrank', 'ext': 'com'}	

36	re flags=re.DOTALL
	multi-line extraction
		txt = """ Extract below code,
		``` # delimiter ``` starts here, to be used in reg expression
		x = 10; y = 20
		print(x+y)
		```
		 # delimiter ends here. **re.DOTALL** enforces **Multi-line search** including (\n).
		That's all!
		"""
		print(re.**findall**("**
		```
		(.*)```**", txt, **flags=re.DOTALL**)) >> **['\nx = 10\ny = 20\nprint(x+y)\n']**	

37	re **Number range** validation	print**(bool(re.match(r'[1-9][0-9]{5}$', num)))** >> validates '**num**' between **100000** - **999999**	
38	re.**findall** - Positive **lookahead**	**Alternative repetitive numbers** find: print(re.**findall**(r'**(\d)(?=\d\1)**', "13717919")) >> **['7', '9']** Here, **\d** → Match and capture a digit in group | **(?=** →Start lookahead | **\d** → Match any digit **\1**: **Back-reference** to **captured group** for searching for **same digit** | **)** → End lookahead	

7. Special Ops

#	Special Operations	Example	Result/Note
1	HTML element attributes extraction (key:value)	html_str = ‘ a **src**=”http www example.com” **alt**=”beautiful mountain” **href**=”http://example.com” ‘ print(re.**findall**(“**\s(.*?)=[\”|\’](.*?)[\”|\’]**”, **html.strip()**)) returns, **[(‘src’, ‘http www example.com’), (‘alt’, ‘beautiful mountain’), (‘href’, ‘http://example.com’)]**	
2	**eval() & exec()** on **string formatted exp** (dynamic code)	lst=[1,2,3]; **eval("print([x**2 for x in "+str(lst)+"])")** >> **[1, 4, 9]** **exec("print([x**2 for x in "+str(lst)+"])")** >> **[1, 4, 9]**	

3	Use of **Zip()** on multiple iterables	A = **[1,2,3,'c']**; B = **[6,5,4,10,11,12,13,14,15]**; C = **[7]**; print(**list(zip(A,B,C))**) >> **[(1, 6, 7)]**
		t1 = "Joydeep"; t2 = "Basu"; print(**list(zip(t1,t2))**) >> **[('J', 'B'), ('o', 'a'), ('y', 's'), ('d', 'u')]**	

4	Use of **exit()** > Stop execution	p = print; p("Hello, world!")
		exit()
		p("Bye bye!")
			Hello, world
			[note: print(Bye bye!”) is unreachable because code stops at exit()]

5	**empty iterable** as looping/if-else condition	a = []
		while a: print(‘Not empty’) >> prints nothing
			while condition is False as a is empty so loop never executes.

6	**Ternary** conditional operation	**print('kid' if age < 18 else 'adult')**
		print('kid' if age < 13 else 'teen' if age < 18 else 'adult')
			<exp1> if <condition> else <exp2>

7	**Overlapping substring** frequency	string’s **count**() method returns **Non-overlapping count** of substring but it **fails for overlapping** substring cases. To overcome this, Use a while **loop** until **ostr.find(sstr)** returns **-1** with original string **slicing[last_match_index+1:]**	If, String: **Banana** and Substring: **ana** **ostr.count(sstr) returns 1** (actual count 2). This can be achieved using below: **while ostr.find(sstr)>=0:** **cnt+=1; ostr=ostr[ostr.find(sstr)+1:]**
8	**Comprehension** without storing results	even, odd, nums = [],[], [1,2,3,4,5,6,7,8,9,44,45,46,47,48,49,100,101,102,103,104,105] **[odd.append(x) if x%2!=0 else even.append(x) for x in nums if x<100]** → *(this result list is not stored)*	
9	Print **all built-ins**	**print(dir(builtins))**	**import builtins** (required to import first)
10	Print all **local scope names**	def func1(): **a=10; s1={1,2,3,5};**	For this function, **print(dir())** >> **[‘a’,’s1’]**
11	Print all **global & local** scope runtime parameters	a,b,c=10,[0,1,2,3,4,5], {1:'Joy', 2:'Deep', 3:'Basu'} print(**globals()**) # **globals()** returns the **dictionary** implementing the **current module namespace**. print(**locals()**) # **locals()** returns a **dictionary** with the **current local symbol table**.	

12	Debugging - use of **breakpoint()**	a = {1:'Joy', 2:[]} ; txt = 'History'
		breakpoint()
		a[2]=txt; print(a)
			Here, **breakpoint() pauses** the execution and enters into debugging mode. Remaining code waits until quit debugging mode.

13	Text message **encoding**	def encode(txt): return print(**bytes**(txt, **encoding='utf-8'**))	Note: use **bytearray** for array of bytes.
14	List element **swapping**	arr = ['Joy','Deep','Basu']; **arr[0], arr[1]** = **arr[1],arr[0]**	
15	Element position (**index**) vs Iterable **index**	lst=[1,2,30,40,90,80,**3**]; print(**lst.index(3),' vs ', lst[3]**) returns **6 vs 40** **lst.index(ele)** returns the **ele** position (index) within lst, whereas, **lst[idx]** returns ele value at idx position.	
16	List item **insertion** using **slicing**	**lst[0:0]=[0]** → (**beginning**) Using **slicing** with same position **index** as **start:end** = [item]**.** **lst[idx:idx] = [50]** → (anywhere in the **middle**) **lst[len(lst):len(lst)]**=[10] → (at the **end**) Note: in all cases new element must be **wrapped** with **[]**.	

17	**Reverse** a Dictionary (value:key) (Note: values can be duplicate so keys with same value must be stored in a list.	res={}
		for key in dct.key():
		    if key not in res: res[key]=[]
		    res[key].append(dct[key])	

18	**Nested Iterable value sorting** (list/tuple of tuple/list, dict of lists) → by value index (idx)	tup=[(1,3,5),(3,2,1),(1,1,1),(5,6,7)] |arr=[[1,3,5],[3,2,1],[1,1,1],[5,6,7]] |dct={1:[1,3,5],2:[3,2,1],3:[1,1,1],4:[5,6,7]} 1. **List of Tuples** sorting by **2nd** value (**idx=1**) → **[x for x in sorted(tup, key = lambda x: x[idx])]** 2. **List of Lists** sorting by **2nd** value (**idx=1**) → **[x for x in sorted(arr, key = lambda x: x[idx])]** 3. **Dict of Lists** sorting by **2nd** value (**idx=1**)>**{k:v for k,v in sorted(dct.items(), key=lambda x: x[1][idx])}**	

19	Collections > **Counter**	**Counter** on string/list/tuple returns,
		Counter('Joy Deep ') >> **Counter({' ': 2, 'e': 2, 'J': 1, 'o': 1, 'y': 1, 'D': 1, 'p': 1})**
		Counter([1,2,3,2,-1,-2,-3]) >> **Counter({2: 2, 1: 1, 3: 1, -1: 1, -2: 1, -3: 1})**
		Counter((1,2,3,2,-1,-2,-3)) >> **Counter({2: 2, 1: 1, 3: 1, -1: 1, -2: 1, -3: 1})**	

20	Collections> **DefaultDictionary**	d = **defaultdict(list)**; print**(d[100])** >> [] # Key 100 is not yet assigned so, default value [] returned. for i in range(1,5): d[i].append(i**2); print(d)>**defaultdict(<class 'list'>, {100: [],1: [1],2: [4],3:[9], 4: [16]})**	
21	Collections > **namedtuple**	**Point = namedtuple('Point', ['x', 'y']); pt = Point(1, 2);** print(**pt.x**, **pt.y**) >> **1 2**	
22	Itertools> **groupby**	txt = "Joyd**ee**p"; lst = [1,2,3,4,**5,5,5**,6,7,8,9,10]; tup = (1,2,3,**4,4**,3,**5,5,5**,6) **[(len(list(g)),k) for k,g in groupby(txt)]** >> [(1, 'J'), (1, 'o'), (1, 'y'), (1, 'd'), (2, 'e'), (1, 'p')] **[(len(list(g)),k) for k,g in groupby(lst)]** >>[(1, 1), (1, 2), (1, 3), (1, 4), **(3, 5),** (1, 6), (1, 7), (1, 8), (1, 9), (1, 10)] **[(len(list(g)),k) for k,g in groupby(tup)]** >> [(1, 1), (1, 2), (1, 3), **(2, 4),** (1, 3), **(3, 5),** (1, 6)]	
23	Itertools > **product**	**list(product(['a','b','c'],repeat=2))** >> *[('a','a'), ('a','b'), ('a','c'), ('b','a'), ('b','b'), ('b','c'), ('c','a'), ('c','b'), ('c','c')]* **list(product([1, 2, 3],repeat=2)) >>** *[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]*	
24	Itertools > **Permutations**	**1. N**-length permutation → **list(permutations([1,2,3],2)) >>** [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)] **2. All possible** permutations**:** [(1,), (2,), (3,), (1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2), (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]	
25	Itertools > **Combinations**(comb) **& combinations with replacement** (cwr)	1. **N-length** combinations: **list(combinations([1,2,3],2))** >> *[(1, 2), (1, 3), (2, 3)]* 2. **All possible** combinations (Powerset): **ps = [] ; for i in range(1, len(arr)+1): ps.extend(list(combinations(arr,i)))** >> *[(1,), (2,), (3,), (1, 2), (1, 3), (2, 3), (1, 2, 3)]* 3. **N-length** combination with replacement: **list(combinations_with_replacement([1,2,3],2))** >> *[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]*	