Day 1

***First Code: Python 3.8***

|  |  |
| --- | --- |
| INPUT CODE | OUTPUT from Console |
| print("Hello")  name = input('What is your name?')  print('Hello ' + name +'.') | Hello  What is your name?Saraj  Hello Saraj.   |

***Fundamental Data Types:***

1. int
2. float
3. bool
4. str
5. list
6. tuple
7. set
8. dict

***Classes: Custom types***

***Specialized Data Types:***  Modules (like an extension to our program)

***None***: none means nothing

Int and Float:

|  |  |
| --- | --- |
| print(2+3)  print(type(2+3))  print(type(1/2))  print(2\*\*3)  print(5/2)  print(5//2) #round off to lower side, floor  print(7%4) | 5  <class 'int'>  <class 'float'>  8  2.5  2  3   |

Math functions:

|  |  |
| --- | --- |
| print(round(3.1))  print(round(3.9))  print(round(3.5))  print(abs(-21)) | 3  4  4  21   |

***Variables***:

* Should start with \_ or small case
* Generally \_ is used for private variables

**Note**: you should not create variables with 2 \_. Magic or Dunder Methods. Dunder methods in Python are the **special methods that start and end with the double underscores**.

Note: following assignment also works

|  |  |
| --- | --- |
| a,b,c=1,2,3  print(a)  print(b)  print(c) | 1  2  3 |

***Augmented assignment operator:***

|  |  |
| --- | --- |
| val = 5  # val = val + 2 #instead of this we can use +=  val += 2  print(val) | 7 |

***Str:***

|  |  |
| --- | --- |
| print(type("Hi"))  # can use single or double quotes  name = 'Saraj'  second\_name = "Sadanand"  #usage of '''  long\_string = '''  WOW  O O  ---  '''  print(long\_string) | <class 'str'>  WOW  O O  ---   |
| first\_name = 'Saraj'  second\_name = "Sadanand"  # *String concatenation (only works with string str)*  name = first\_name + ' ' + second\_name  print(name) | Saraj Sadanand   |

***Type conversion:***

|  |  |
| --- | --- |
| print(type(str(100)))  print(type(int(str(100)))) | <class 'str'>  <class 'int'>   |

***Escape sequence****, using a backslash****:***

|  |  |
| --- | --- |
| weather = "It's \"kind of\" sunny today."  print(weather) | It's "kind of" sunny today.   |

Similarly, \t for tab, \n for newline

***Formatted Strings or F String:***

|  |  |
| --- | --- |
| name = 'Saraj'  age = 55  print('Hi ' + name + '. You are ' + str(age) + ' years old.')  # in Python 2  print('Hola {0}. You are {1} years old.'.format(name,age))  # formatted strings, Python 3 new feature  print(f'Hello {name}. You are {age} years old.') | Hi Saraj. You are 55 years old.  Hola Saraj. You are 55 years old.  Hello Saraj. You are 55 years old.   |

***String indexes:***

|  |  |
| --- | --- |
| line = 'Hi buddy whats up'  # 01234567890123456  print(line[3])  print(line[3:7]) #7 excluded  test = '01234567890123456789'  print(test[0:15:2]) # [start:stop:stepover] | b  budd  02468024   |

* To reverse a string:

|  |  |
| --- | --- |
| test = '0123456789'  print(test[::-1]) # [start:stop:stepover]  print(test[::-2]) | 9876543210  97531   |

Note:

* Strings are immutable. We cannot edit them, we can only recreate.
* E.g. name = 'saraj'

Now you cannot do 🡪 name[2] = 'a' # incorrect

But, this will work 🡪 name = 'sai'

***Built-In Functions and Methods:***

* print() is a function whereas .find(), .replace() are methods, i.e. it is a str method. Methods are always with a dot.

|  |  |
| --- | --- |
| quote = "to be or not to be"  print(quote.upper()) #upper is a method | TO BE OR NOT TO BE   |

***Booleans:***  “False” or “True”

print(bool(0)) 🡪 False

print(bool(1)) 🡪 True

Day 2

Exercise:

|  |  |
| --- | --- |
| # Type conversion  birth\_year = input("What year were you born?\n")  age = 2022 - int(birth\_year)  print(f"You are {age} years old.") | What year were you born?  1993  You are 29 years old.   |
| # Password Checker  username = input("Enter the username: ")  password = input("Enter the password: ")  pass\_len = len(password)  print(f"Hi {username}, your password, {'\*' \* pass\_len}, is {pass\_len} letters long.") | Enter the username: saraj  Enter the password: secretcode  Hi saraj, your password, \*\*\*\*\*\*\*\*\*\*, is 10 letters long.   |

**DATA SRTUCTURES (list, dict, tuples, set)**

***Lists:***

|  |  |
| --- | --- |
| amazon\_cart = [  "notebook",  "sunglass",  'toys',  'grapes',  ]  print(amazon\_cart)  # lists are mutable  amazon\_cart[0] = "laptop"  print(amazon\_cart)  # list slicing  new\_amazon\_cart = amazon\_cart[0:2] # items 0,1  print(new\_amazon\_cart) | ['notebook', 'sunglass', 'toys', 'grapes']  ['laptop', 'sunglass', 'toys', 'grapes']  ['laptop', 'sunglass']   |

|  |  |
| --- | --- |
| # IMPORTANT  amazon\_cart = [  "notebook",  "sunglass",  'toys',  'grapes',  ]  print(f'{amazon\_cart}\n\n')  new\_amazon\_cart = amazon\_cart **# NOTE: This will use the same memory space**  new\_amazon\_cart[0] = 'laptop'  print(new\_amazon\_cart)  print(amazon\_cart) **# the original list is also replaced**  print('\n')  # SO ALWAYS COPY A LIST AS SHOWN BELOW  newer\_amazon\_cart = amazon\_cart[:] # this will create a new list  newer\_amazon\_cart[0] = 'desktop'  print(newer\_amazon\_cart)  print(amazon\_cart) | ['notebook', 'sunglass', 'toys', 'grapes']  ['laptop', 'sunglass', 'toys', 'grapes']  ['laptop', 'sunglass', 'toys', 'grapes']  ['desktop', 'sunglass', 'toys', 'grapes']  ['laptop', 'sunglass', 'toys', 'grapes']   |

*Matrix:*

* lists inside a list, like a 2D list

matrix = [

[12,34,2],

[2,5,7],

[121,5,21]

]

# The above matrix has 3 items in the list and each item is again a list.

List methods:

|  |  |
| --- | --- |
| list = [1,2,3,4,5]  # the below methods change the list "inplace"  new\_list = list.append(100)  print(new\_list) # returns None  print(list)  print('------------------')  ##### for adding items to the list  # correct approach  list = [1,2,3,4,5]  list.append(100)  new\_list = list  print(new\_list)  list.insert(2,22) # at index 2 the value is inserted  print(list)  list.extend([33,44])  print(list)  ##### for removing items from the list  list.pop()  print(list)  list.pop(2) # with pop we give index  print(list) # pop returns the popped value  list.remove(100) # with remove we give value  print(list)  list.clear()  print(list) | None  [1, 2, 3, 4, 5, 100]  ------------------  [1, 2, 3, 4, 5, 100]  [1, 2, 22, 3, 4, 5, 100]  [1, 2, 22, 3, 4, 5, 100, 33, 44]  [1, 2, 22, 3, 4, 5, 100, 33]  [1, 2, 3, 4, 5, 100, 33]  [1, 2, 3, 4, 5, 33]  []   |

Day 3

|  |  |
| --- | --- |
| basket = ['a','x','b','c','d','e','d']  basket1 = basket[:]  basket2 = basket.copy() # same as above but using the copy method  print('d' in basket)  print('i' in "Hi my name is blah")  print(basket.count('d'))  # print(basket.sort()) # doesnt produce anything as the change is inplace  basket.sort() # sort method is inplace  print(basket)  # sorted function is also there  print(sorted(basket1))  print(basket1) # which means the sorthed function doesn't change inplace  print("----------------")  # To reverse sort a list  basket2.sort()  basket2.reverse() # inplace change, since sort and reverse are methods  print(basket2)  #OR  print(basket2[::-1])  print(basket2) | True  True  2  ['a', 'b', 'c', 'd', 'd', 'e', 'x']  ['a', 'b', 'c', 'd', 'd', 'e', 'x']  ['a', 'x', 'b', 'c', 'd', 'e', 'd']  ----------------  ['x', 'e', 'd', 'd', 'c', 'b', 'a']  ['a', 'b', 'c', 'd', 'd', 'e', 'x']  ['x', 'e', 'd', 'd', 'c', 'b', 'a']   |

Some common list patterns:

|  |  |
| --- | --- |
| # range function  print(list(range(1,101))) | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100] |
| # join meathod  sentance = "Hi"  list = ["Saraj", "my", "name", "hello"]  new\_sentance = sentance.join(list)  print(new\_sentance)  -------------------------------------------------------------------  ***# Used if you want to join and create a new string from a list:***  space = " "  list1 = ["My", "name", "is", "Saraj."]  print(list1)  sentance = space.join(list)  print(sentance)  **#OR**  sentence = " ".join(list) | SarajHimyHinameHihello    -------------------------------------------------------------------  ['My', 'name', 'is', 'Saraj.']  My name is Saraj.   |

List Unpacking:

|  |  |
| --- | --- |
| # Usually we can do a,b,c = 1,2,3 also below works the same way  # a,b,c = [1,2,3]  a,b,c, \*other = [1,2,3,4,5,6,7,8,9]  print(a)  print(b)  print(c)  print(other)  print("---------------------")  a,b,c, \*other, d = [1,2,3,4,5,6,7,8,9]  print(a)  print(b)  print(c)  print(other)  print(d) | 1  2  3  [4, 5, 6, 7, 8, 9]  ---------------------  1  2  3  [4, 5, 6, 7, 8]  9   |

None:

* special datatype
* represents absence of value
* we can do, a = None, in the code. e.g. in a game, weapons = None

Day 4

***Dictionary***: unordered key-value pair

* keys cannot change i.e. it is immutable like str
* a dict key always has to be immutable, so as a key we cannot use list
* as a key we can use int, str, bool but not a list
* tuple is a valid key in dict
* a key in a dict has to be unique, no error is thrown if keys are repeated
* if same key is repeated, the last value is overwritten

|  |  |
| --- | --- |
| dict0 = {  'a' : 1,  'b' : 2  }  print("----------")  print(dict0)  dict1 = {  'a' : [1,2,3], # list  'b' : "hello", # str  'c' : True # bool  }  print(dict1)  print(dict1['a'])  print(dict1['a'][1]) | {'a': 1, 'b': 2}  ----------  {'a': [1, 2, 3], 'b': 'hello', 'c': True}  [1, 2, 3]  2   |

* A list can contain dict
* How to access values withing a dict which is present in a list?

|  |  |
| --- | --- |
| my\_list = [  {  'a' : [1,2,3], # list  'b' : "hello", # str  'c' : True # bool  },  {  'p' : [4,55,6],  'q' : 'hi'  }  ]  # if you want to access 55  print(my\_list[1]['p'][1]) | 55 |

Insert item in a dict:

|  |  |
| --- | --- |
| thisdict = {  "brand": "Ford",  "model": "Mustang",  "year": 1964  }  thisdict["color"] = "red"  print(thisdict) | {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'} |

.get() method in dictionary:

|  |  |
| --- | --- |
| # to check if a key exists, we have .get method  user = {  'basket' : [1,2,3],  'greet' : "hello"  }  print(user.get('greet')) | hello |
| user = {  'basket' : [1,2,3],  'greet' : "hello"  }  print(user.get('age')) | None |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  }  print(user.get('age',18)) # if age does not exist, take 18 as default value | 18 |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  print(user.get('age',18)) # age exists, so does not take 18 | 22 |

FYI, dict() is a built in function which is used as an alternative to create a dictionary (rarely used):

|  |  |
| --- | --- |
| user = dict(name='Saraj')  print(user) | {'name': 'Saraj'}   |

Some more dict methods:

|  |  |
| --- | --- |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  print('age' in user)  print('height' in user) | True  False   |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  print('age' in user.keys())  print('hello' in user.values())  print(2 in user.values()) | True  True  False   |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  print(user.items()) | dict\_items([('basket', [1, 2, 3]), ('greet', 'hello'), ('age', 22)])    # this has a key value pair as a tuple |

|  |  |
| --- | --- |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  print(user.clear()) | None   |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  user.clear()  print(user) | {}   |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  user2 = user.copy()  print(user2) | {'basket': [1, 2, 3], 'greet': 'hello', 'age': 22}   |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  print(user.pop('greet')) # pop returns the value of whatever being removed  print("--------------")  print(user) | hello  --------------  {'basket': [1, 2, 3], 'age': 22}   |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  print(user.popitem()) # pops the last inserted item  print(user) | ('age', 22)  {'basket': [1, 2, 3], 'greet': 'hello'}   |
| user = {  'basket' : [1,2,3],  'greet' : "hello",  'age' : 22  }  user.update({'age' : 55})  print(user)  user.update({'ages' : 65}) # if update keu doesn't exist, then new item is inserted  print(user) | {'basket': [1, 2, 3], 'greet': 'hello', 'age': 55}  {'basket': [1, 2, 3], 'greet': 'hello', 'age': 55, 'ages': 65}   |

***Tuple:***

* just an immutable list
* less flexible, cannot sort or reverse

|  |  |
| --- | --- |
| my\_tuple = (1,2,3,4,5)  print(my\_tuple)  print(my\_tuple[1])  print("-----------------")  my\_tuple[1] = 'z' # not allowed | (1, 2, 3, 4, 5)  2  -----------------  Traceback (most recent call last):  File "main.py", line 5, in <module>  my\_tuple[1] = 'z'  TypeError: 'tuple' object does not support item assignment   |
| my\_tuple = (1,2,3,4,5)  print(5 in my\_tuple) | True |
| user = {  'basket' : [1,2,3],  (1,2) : "hello", # using a tuple as a key  'age' : 22  }  print(user)  print(user['basket'])  print(user[(1,2)]) | {'basket': [1, 2, 3], (1, 2): 'hello', 'age': 22}  [1, 2, 3]  hello   |

* tuple has just two methods

1. count()
2. index()

|  |  |
| --- | --- |
| my\_tuple = (1,2,3,4,5,2,2)  print(my\_tuple.count(2))  print(my\_tuple.index(5))  print(my\_tuple.index(2)) # returns the index of first value it finds  print(f"Length from len function is: {len(my\_tuple)}") | 3  4  1  Length from len function is: 7   |

***Set***:

* used to store multiple items in a single variable, unordered collection of unique objects
* set does not support indexing, like print(my\_set[1]) will not work

|  |  |
| --- | --- |
| my\_set = {2,1,2,3,4,5,5,3}  print(my\_set) # stores only unique values, unordered  print(len(my\_set) | {1, 2, 3, 4, 5}  5   |

**Question**: From a list, fetch only the unique values.

*Using the list() and set() functions:*

|  |  |
| --- | --- |
| my\_list = [2,1,2,3,4,5,5,3]  my\_set = set(my\_list)  print(my\_set)  my\_list\_new = list(my\_set)  print(my\_list\_new) | {1, 2, 3, 4, 5}  [1, 2, 3, 4, 5]   |
| my\_list = [2,1,2,3,4,5,5,3]  print(list(set(my\_list))) | [1, 2, 3, 4, 5]   |

***Program*** *to get unique values from a list:*

|  |  |
| --- | --- |
| # function to get unique values  def unique(list):  unique\_list = [] # initializing a null list  for i in list:  if i not in unique\_list:  unique\_list.append(i)  print(unique\_list)  # driver code  my\_list = [1,2,3,4,2,3,4,5]  unique(my\_list) | [1, 2, 3, 4, 5]   |

set methods: <https://www.w3schools.com/python/python_ref_set.asp>

|  |  |
| --- | --- |
| set\_a = {1,2,3,4,5}  set\_b = {4,5,6,7,8}  print(set\_a.difference(set\_b))  print(set\_a.discard(5)) # changes inplace hence returns None  print(set\_a) | {1, 2, 3}  None  {1, 2, 3, 4}   |
| set\_a = {1,2,3,4,5}  set\_b = {4,5,6,7,8}  set\_a.difference\_update(set\_b)  print(set\_a) | {1, 2, 3} |
| set\_a = {1,2,3,4,5}  set\_b = {4,5,6,7,8}  print(set\_a.intersection(set\_b)) | {4, 5} |

***Conditional Logic:***

|  |
| --- |
| is\_old = True  is\_licenced = False  if is\_licenced:  print("You can drive.")  elif is\_old:  print("You can apply for licence and then drive.")  .  .  .  elif <condition>:  .  .  .  else:  print("You cannot drive.") |

***Truthy and Falsy:***

True: bool(“hello”), bool(5)

False: bool(“”), bool(0)

|  |  |
| --- | --- |
| username = 'saraj'  password = 'superpassword'  # we can use directly like the below usage as the variables return truthy/falsy values  if username and password:  print("You have entered username and password") | You have entered username and password   |

***Terenary Operator:***

|  |  |
| --- | --- |
| # Terenary operator:  # condition\_if\_true if condition else condition\_if\_false  is\_friend = True  can\_message = "message allowed" if is\_friend else "not allowed to message"  print(can\_message) | message allowed   |

***Logical Operator:***

* and
* or
* <
* >
* == [**Note**, = is assignment used in an expression, which will work only on variables that start with alphabet or \_ ]
* >=
* <=

|  |  |
| --- | --- |
| is\_friend = False  is\_user = True  print(is\_friend and is\_user)  if is\_friend or is\_user:  print('Best friends.')  print(5>3>4>1) # gives false as one condition is false, short circuting | False  Best friends.  False   |

Day 5

Some more usage:

|  |
| --- |
| if is\_expert and is\_magician |
| elif is\_magician and not is\_expert # not will just invert whatever the Boolean expression is |
| elif not is\_magician # no need to end with else |

***is vs ==***

* is compares what is placed in a particular memory location
* for lists, because it is a data structure, everytime we create a list although with same value the ‘is’ will give false

|  |  |
| --- | --- |
| print(True == 1) # 1 means bool(1) which means true  print('' == 1) # empty stmt is falsy  print([] == 1) # empty array s falsy  print(10.0 == 10)  print([] == [])  print('1' == 1) # comparing two types | True  False  False  True  True  False |
| print(True is 1)  print('' is 1)  print([] is 1)  print(10.0 is 10)  print([] is [])  print('1' is 1) | False  False  False  False  False  False |
| print(True is True)  print('1' is '1')  print(10 is 10)  print([] is []) # every DS is assigned with a new mem space  a = [1,2,3]  b = [1,2,3]  print(a == b) # comparing the values | True  True  True  False  True   |

Day 6

***Looping***

***for***

|  |  |
| --- | --- |
| for i in 'how are you?':  print(i)  # for sets  for item in {1,2,3}:  print(item)  # for lists  for item in [111,222,333,444]:  print(item)  # for tuple  for i in (1,2,3,4):  print(i) | h  o  w    a  r  e    y  o  u  ?  1  2  3  111  222  333  444  1  2  3  4   |
| # nested loop  for item in (1,2,3,4):  for x in ['a','b','c']:  print(item, x) | 1 a  1 b  1 c  2 a  2 b  2 c  3 a  3 b  3 c  4 a  4 b  4 c   |

**Iterable:** list, dict, tuple, set, string 🡪 one by one check each item in the collection

* int is not iterable, so “for i in 50” will not work

|  |  |
| --- | --- |
| # dict  user = {  'name': "Saraj",  'age': 55,  'can\_swim': False  }  for i in user:  print(i) # prints the keys  # there are three important methods  for i in user.items():  print(i)  for i in user.keys():  print(i)  for i in user.values():  print(i) | name  age  can\_swim  -------------------  ('name', 'Saraj')  ('age', 55)  ('can\_swim', False)  name  age  can\_swim  Saraj  55  False   |
| # common usage  user = {  'name': "Saraj",  'age': 55,  'can\_swim': False  }  for key,value in user.items():  print(key, value) | name Saraj  age 55  can\_swim False   |

*Question*: Using looping, sum up the list.

|  |  |
| --- | --- |
| my\_list = [1,2,3,4,5]  sum = 0  for i in my\_list:  sum += i  print(f"The sum is {sum}.") | The sum is 15.   |

***range() function:***

|  |  |
| --- | --- |
| for num in range(0,5):  print(num) | 0  1  2  3  4   |
| # if the variable in for (num in above e.g.) is not to be used in the loop anywhere then use \_  for \_ in range(0,5):  print("Email sending") | Email sending  Email sending  Email sending  Email sending  Email sending   |
| for \_ in range(0,10,2):  print(\_) | 0  2  4  6  8   |
| for \_ in range(5,0,-1):  print(\_) | 5  4  3  2  1   |
| for \_ in range(2): #will run loop for \_ values 0 and 1  print(list(range(5))) | [0, 1, 2, 3, 4]  [0, 1, 2, 3, 4]   |

***enumerate() function:***

|  |  |
| --- | --- |
| for i,letter in enumerate("hello"): #this assigns an index value to i  print(i, letter) | 0 h  1 e  2 l  3 l  4 o   |

Question: Find the index of an item from a list.

|  |  |
| --- | --- |
| for i, num in enumerate(list(range(101,201))):  if num == 155:  print(f"Index of 155 is {i}.") | Index of 155 is 54.   |

Day 7

***While loop***

|  |  |
| --- | --- |
| i = 0  # below will be an infinite loop  #while i < 50:  # print(i)  while i < 50:  print(i)  break # prints 0 once and comes out of the loop  # print 0-10  while i < 10:  print(i)  i += 1  #break # this will break out of while loop and else is also ignored  else:  print("Done with printing...")  # else will be skipped if there is 'break' in while | 0  0  1  2  3  4  5  6  7  8  9  Done with printing...   |
| *# Printing a list using for and while*  my\_list = [11,22,33]  for item in my\_list:  print(item)  i = 0  while i < len(my\_list):  print(my\_list[i])  i += 1 | 11  22  33  11  22  33   |

Common usage of while:

|  |  |
| --- | --- |
| **while True:**  **response = input("Say something: ")**  **if response == "bye" or response == "Bye":**  **break** | Say something: hi  Say something: hello  Say something: how are you  Say something: Bye       |

***break, continue, pass:***

break - breaks out of the enclosing loop

continue - goes back to the start of enclosing loop

pass - just to skip, like a place holder (rare in prod, sometimes used during the development)

|  |  |
| --- | --- |
| my\_list = [11,22,33]  for item in my\_list:  # still thinking about the logic  pass  i = 0  while i < len(my\_list):  print(my\_list[i])  i += 1 | If pass is not used, below error appears in the o/p  File "main.py", line 6  i = 0  ^  IndentationError: expected an indented block   |

|  |  |
| --- | --- |
| #Exercise!  #Display the image below to the right hand side where the 0 is going to be ' ', and the 1 is going to be '\*'. This will reveal an image!  picture = [  [0,0,0,1,0,0,0],  [0,0,1,1,1,0,0],  [0,1,1,1,1,1,0],  [1,1,1,1,1,1,1],  [0,0,0,1,0,0,0],  [0,0,0,1,0,0,0]  ]  for row in picture:  for i in row:  if i == 1:  print('\*', end = '')  else:  print(' ', end = '')  print('') # just to insert a new line after every row | Hint: |

Exercise: Check duplicate values from a list

|  |  |
| --- | --- |
| ls = ['a', 'b', 'c','b','a','d','e','d','f'] duplicates = [] for i in ls:  if ls.count(i) > 1:  if i not in duplicates:  duplicates.append(i)   print(duplicates) | ['a', 'b', 'd'] |

***Functions:***

|  |  |
| --- | --- |
| def say\_hello():  print("Hello World...!")  say\_hello() # function call  print(say\_hello) # gives memory address | Hello World...!  <function say\_hello at 0x7f566261c280>   |

***Parameters and Arguments:***

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that are sent to the function when it is called.

|  |  |
| --- | --- |
| # parameters  def say\_hello(name,emoji):  print(f"Hello {name} {emoji}")  # arguments, positional arguments  say\_hello('Saraj', '🙂')  say\_hello('Sai','🤓')  # keyword arguments (not a good practice)  say\_hello(emoji = '🙂', name = 'Sangi') | Hello Saraj 🙂  Hello Sai 🤓  Hello Sangi 🙂   |

***Default Parameters:***

|  |  |
| --- | --- |
| # default parameters  def say\_hello(name = 'Angry Man',emoji = '😈'):  print(f"Hello {name} {emoji}")  say\_hello()  say\_hello('Saraj', '🙂')  say\_hello('Saraj') # second argument will be default | Hello Angry Man 😈  Hello Saraj 🙂  Hello Saraj 😈   |

***Return:***

|  |  |
| --- | --- |
| def sum(num1, num2):  print("Hi...")  num1 + num2  print(sum(4,5)) # gives None | Hi...  None   |
| def sum(num1, num2):  return num1 + num2 # returns the result of this expression instead of none  print(sum(10,5)) | 15   |

***Note***: return exits the function, so any line after return will not be run

Functions vs Methods:

Functions:

* list(), print(), max(), min(), input() etc.
* def some\_random\_function()

pass

some\_random\_function()

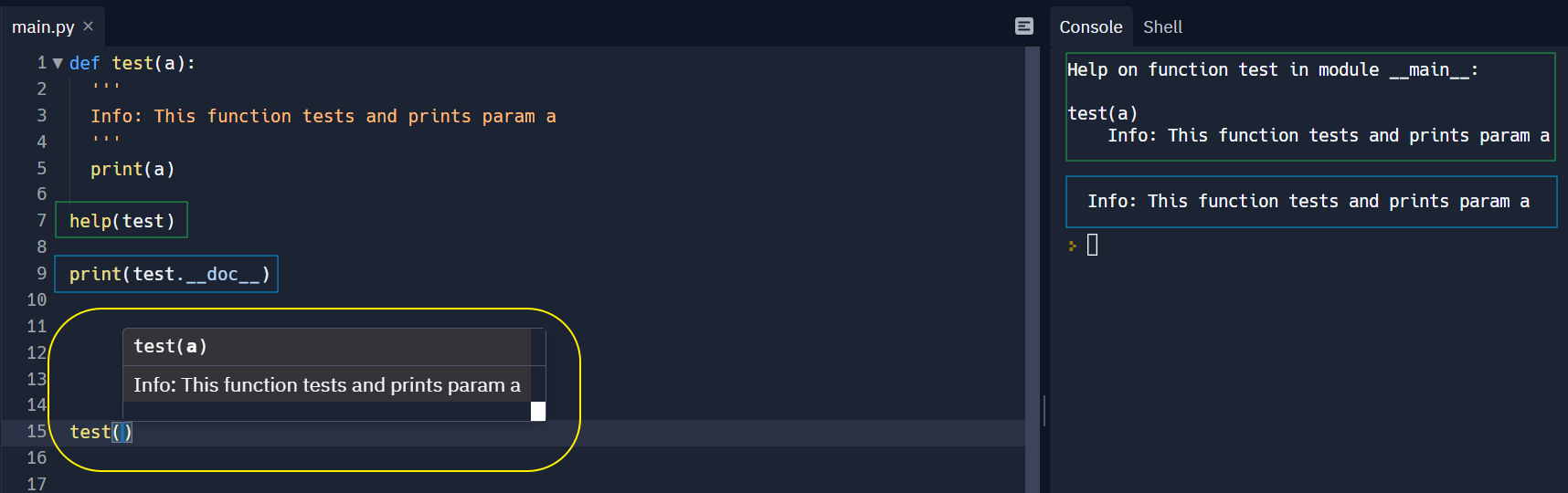
Methods:

* has to be owned by something, whatever is left to the dot
* ‘hello’.capitalize() 🡪 here the capitalize method is owned by string ‘hello’

***Docstrings:***

A Python docstring is **a string used to document a Python module, class, function or method**, so programmers can understand what it does without having to read the details of the implementation. Also, it is a common practice to generate online (html) documentation automatically from docstrings.

Should be enclosed within ''' and '''. Below, at line 15 when we try to call the function, the editor smartly displays the docstring provided in the function definition.



***\*args and \*\*kwargs***

* We use \*args OR \*\*kwargs as our function’s parameter when we have doubts about the number of arguments we should pass in a function.
* We can use any name instead of \*args OR \*\*kwargs, however this is the standard way for developers to understand.
* *Order of arguments as a rule*: **params, \*args, default\_params, \*\*kwargs**

|  |  |
| --- | --- |
| def super\_func(\*args):  print(\*args)  print(args) # prints the arguments as a tuple  return sum(args) # sum is an inbuild function  print(super\_func(1,2,3,4,5)) | 1 2 3 4 5  (1, 2, 3, 4, 5)  15   |
| def super\_func(\*args, \*\*kwargs):  print(args) # args as a tuple  print(kwargs) # this returns keyword arguments as a dict  total = 0  for i in kwargs.values():  total += i  return sum(args) + total   print(super\_func(1, 2, 3, 4, 5, num1=10, num2=20)) | (1, 2, 3, 4, 5)  {'num1': 10, 'num2': 20}  45 |

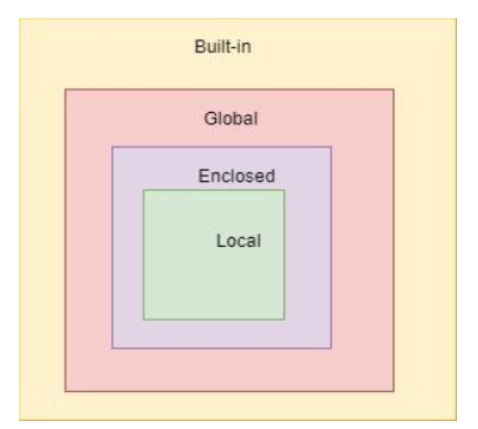
Question: Find the highest even number from a list.

|  |  |
| --- | --- |
| def highest\_even(ls):  ls\_even = []  for item in ls:  if item % 2 == 0:  ls\_even.append(item)  return max(ls\_even)  print(highest\_even([10,2,3,4,8,11])) | 10   |

**Walrus Operator :=**

|  |  |
| --- | --- |
| a = 'Hellooooooooooooooooooooooooooo'  # if len(a) > 5:  # print(f"Too many charancters, a total of {len(a)}")  # this is usual method  if (n := len(a)) > 5:  print(f"Too many charancters, a total of {n}")  # Here’s instead of using “len(a)” in two places we have assigned it to a variable called “n”, which can be used later. This helps us to resolve code duplication and improves readability. | Too many charancters, a total of 31   |

**Scope Rules:**



***global*** **keyword**

|  |  |
| --- | --- |
| total = 0  def count():  global total # to access the global variable total at line 1  total += 1  return total  count()  count()  print(count()) | 3 |

***nonlocal* keyword**

|  |  |
| --- | --- |
| def outer\_func():  x = 'local'  def inner\_func():  nonlocal x # accesses the x of the enclosed function  x = 'inner'  print("Inner:" +x)  inner\_func()  print("Outer:" +x)  outer\_func() | Inner:inner  Outer:inner   |

Day 9

***OOP***

**Class:** is like a blueprint

**Object:** using the class we can create (instantiate) instances also called objects

|  |  |
| --- | --- |
| class BigObject:  # code  pass   ob1 = BigObject() # instantiate (creating an object) ob2 = BigObject() # instantiate ob3 = BigObject() # instantiate print(type(BigObject())) | <class '\_\_main\_\_.BigObject'> |

|  |  |
| --- | --- |
| class PlayerCharacter:  def \_\_init\_\_(*self*, name, age): # dunder method  *self*.name = name # attributes  *self*.age = age   def run(*self*): # method  print('Running...')   player1 = PlayerCharacter("Saraj", 22) player2 = PlayerCharacter("Sai", 11) print(player1.name, player2.age) print(player1.run()) player2.attack = 50 # added a new attribute print(player2.attack) | Saraj 11  Running...  None 🡪 since the function run() returns nothing  50 |

**Note about *\_\_init\_\_*:**

* The \_\_init\_\_ function is called every time an object is created from a class. It is run as soon as an object of a class is instantiated.
* The \_\_init\_\_ method lets the class initialize the object's attributes and serves no other purpose.
* It is only used within classes. he \_\_init\_\_ method is similar to **constructors**in C++ and Java. Constructors are used to initialize the object’s state.

**Important Code:**

|  |
| --- |
| class PlayerCharacter:  # Class Object Attribute  membership = True   def \_\_init\_\_(*self*, name, age):  if PlayerCharacter.membership: # also we can give self.membership  *self*.name = name # attributes  *self*.age = age   def shout(*self*):  print(f'My name is {*self*.name}!')  return '...'   def run(*self*, speed):  print(f"{*self*.name} is running at {speed}")   player1 = PlayerCharacter("Saraj", 22) player2 = PlayerCharacter("Sai", 11) player2.attack = 50 # added an additional attribute print(player1.shout()) print(player2.membership) print(PlayerCharacter.membership) # Class object attribute can be accessed directly without an object also print(player1.run("16km/h")) # since the method speed() is expecting an argument speed print(player2.run("12km/h")) # also note that this method does not have a return |
| My name is Saraj!  ...  True  True  Saraj is running at 16km/h  None  Sai is running at 12km/h  None |

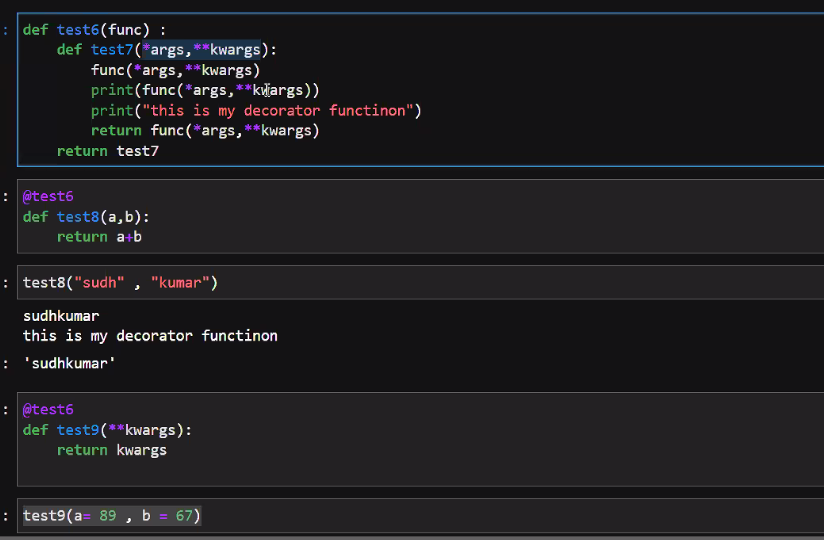
Exercise:

|  |
| --- |
| #Given the below class:  class Cat:  species = 'mammal'  def \_\_init\_\_(self, name, age):  self.name = name  self.age = age  # 1 Instantiate the Cat object with 3 cats  # 2 Create a function that finds the oldest cat  # 3 Print out: "The oldest cat is x years old.". x will be the oldest cat age by using the function in #2 |
| class Cat:  species = 'mammal'   def \_\_init\_\_(*self*, name, age):  *self*.name = name  *self*.age = age   # Instantiate the Cat object with 3 cats peanut = Cat("Peanut", 3) garfield = Cat("Garfield", 5) snickers = Cat("Snickers", 1)   # Find the oldest cat def get\_oldest\_cat(\*args):  return max(args)   # Output print(f"The oldest cat is {get\_oldest\_cat(peanut.age, garfield.age, snickers.age)} years old.") |

**Decorator: @test is a decorator**



**Always give \*args and \*\*kwargs and give return as below:**

****

|  |
| --- |
| # Decorator e.g.  def deco(func):  def test(\*args,\*\*kwargs):  print("Test starts...")  func(\*args,\*\*kwargs)  print("...Test ends!")  return func(\*args,\*\*kwargs)  return test  @deco  def test\_new(a,b):  print("Inside test\_new function"+a+b)  test\_new("saraj","abc")  *O/P:*  Test starts...  Inside test\_new functionsarajabc  ...Test ends!  Inside test\_new functionsarajabc |

**@classmethod 🡺** This method can be called without instantiaing an object of the class.

You can use class methods for **any methods that are not bound to a specific instance but the class**.

|  |
| --- |
| class PlayerCharacter:  # Class Object Attribute  membership = True   def \_\_init\_\_(*self*, name, age):  if PlayerCharacter.membership: # also we can give slef.membership  *self*.name = name # attributes  *self*.age = age   def run(*self*, speed):  print(f"{*self*.name} is running at {speed}")   @classmethod  def adding\_things(*cls*, num1, num2):  return num1 + num2  # instead of above we can also do  # return cls('Sooraj', num1 + num2) # with this we can directly instantiate a new name called Sooraj with age 5   print(PlayerCharacter.adding\_things(2, 3))  Output: 5 |
| class PlayerCharacter:  # Class Object Attribute  membership = True   def \_\_init\_\_(*self*, name, age):  if PlayerCharacter.membership: # also we can give slef.membership  *self*.name = name # attributes  *self*.age = age   def run(*self*, speed):  print(f"{*self*.name} is running at {speed}")   @classmethod  def adding\_things(*cls*, num1, num2):  return *cls*('Sooraj', num1 + num2) # with this we can directly instantiate a new name called Sooraj with age 5   player3 = PlayerCharacter.adding\_things(2, 3) print(player3.name, player3.age)  Output: Sooraj 5 |

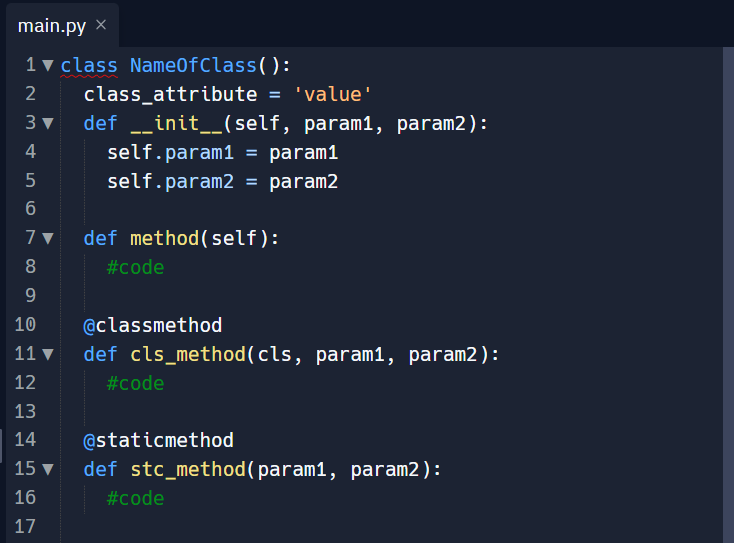
**@staticmethod**:

**A class method takes cls as the first parameter while a static method needs no specific parameters**. A class method can access or modify the class state while a static method can't access or modify it. In general, static methods know nothing about the class state.

|  |
| --- |
| @classmethod def adding\_things(*cls*, num1, num2):  return *cls*('Sooraj', num1 + num2) # with this we can directly instantiate a new name called Sooraj with age 5  @staticmethod def adding\_things2(num1, num2):  return num1 + num2 |

<https://stackoverflow.com/questions/136097/difference-between-staticmethod-and-classmethod>

***Whole idea until now is:***



**4 PILLARS OF OOP**

***Encapsulation***

class 🡺 attributes (within *\_\_init\_\_* method) + methods

***Abstraction***

The idea of private variable/attribute (like in Java) is not there in Python, we just give an \_ as a convention to denote a private variable.

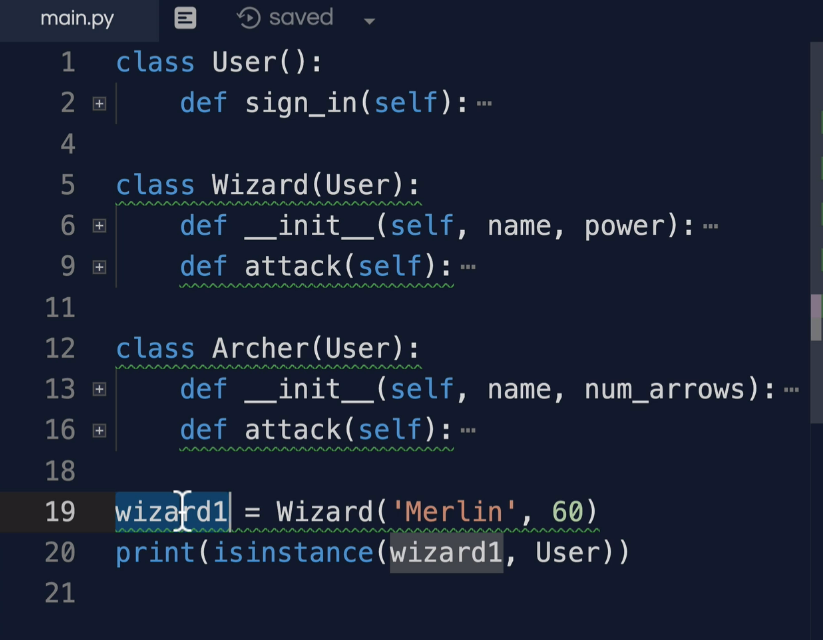
class PlayerCharacter:  
 def \_\_init\_\_(*self*, name, age):  
 *self*.\_name = name  
 *self*.\_age = age

***Inheritance***

User 🡨 Wizard 🡨 Archer

|  |
| --- |
| class User():  def sign\_in(*self*):  print("Logged in")   class Wizard(User):  pass   class Archer(Wizard):  pass   player = Wizard() print(player) print(player.sign\_in()) |
| <\_\_main\_\_.Wizard object at 0x0000021EE3D97EB0>  Logged in |

**isinstance:** The isinstance() function returns True if the specified object (wizard1) from the specified class (User), otherwise False.



In python, everything is an object. All objects in python inherit from a base class called **object**. So, instead of User if we give object even then isinstance will return true. \_\_init\_\_(), \_\_dir\_\_() etc. are all part of the object class.

***Polymorphism:***

Same method names are used to behave in different ways.

***NOTEPAD:***

***PYTHON: OOP CONCEPT CODE***

***` go through the udemy videos***

***` write the code and practice***

***` make notes***

***Functional Programming:***

*PURE FUNCTION*: A pure function is a function that will return the same values given the same arguments. A function is not pure if there is something outside the function that can change its return, given the same arguments.

***Python built-in functions:***

***map, filter, reduce and zip***

<https://www.analyticsvidhya.com/blog/2021/07/python-most-powerful-functions-map-filter-and-reduce-in-5-minutes/>

map()

|  |
| --- |
| # map  def multiplyby2(li):  new\_ls = []  for i in li:  new\_ls = new\_ls.append(i \* 2)  return new\_ls   # instead of above, just write the below code and use map   def multiply\_by2(item):  return item \* 2   print(list(map(multiply\_by2, [1, 2, 3]))) # map will run the function and loop all the items in the iterable part |
| [2, 4, 6] |

filter()

|  |
| --- |
| # filter  my\_list = [1,2,3,4,5]  def multiply\_by2(item):  return item \* 2   def only\_odd(item):  return item % 2 != 0   print(list(filter(only\_odd, my\_list))) |
| [1, 3, 5] |

reduce()

|  |  |
| --- | --- |
| from functools import reduce  my\_list = [1,2,3]   def accumulator(acc, item):  print(acc, item)  return acc + item   print(reduce(accumulator, my\_list, 0)) # acc will start with 0 print(reduce(accumulator, my\_list, 10)) # acc initialised with 10 | 0 1  1 2  3 3  6  10 1  11 2  13 3  16 |

zip()

|  |
| --- |
| my\_list = [1,2,3] your\_list = [11,22,33]  print(list(zip(my\_list,your\_list))) # zipped as a tuple |
| [(1, 11), (2, 22), (3, 33)] |

we can give more that 2 lists/tuples in the zip function which will zip all of them together.

**Lambda expression:**

Functions with no name are known as lambda functions. These functions are frequently used as input to other functions. Let’s try to integrate Lambda functions into the map() function.

EXAMPLE

tup= (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)

newtuple = tuple(map(lambda x: x+3 , tup))

print(newtuple)

OUTPUT

(8, 10, 25, 100, 57, 65, 80, 26, 76, 64)

Q. Using lambda, find the square of a list containing numbers.

# square  
my\_list = [5,6,7]  
sq\_list = map(lambda x: x\*\*2, my\_list)  
print(list(sq\_list))

[25, 36, 49]

Q. List sorting based on second item in the tuple.

# List sorting based on second item in the tuple inside of a list  
a = [(0,2), (4,3), (10,-1), (9,9)]  
a.sort  
print(a) # this has sorted the list based on first item in the tuple  
a.sort(key=lambda x: x[1])  
print(a)

[(0, 2), (4, 3), (10, -1), (9, 9)]

[(10, -1), (0, 2), (4, 3), (9, 9)]

*This part is important:*

**List comprehensions:** Ways to quickly create lists without writing loops or functions.

Q. To change a string to character list.

my\_list = []  
for char in 'hello':  
 my\_list.append(char)  
print(my\_list)

['h', 'e', 'l', 'l', 'o']

Alternatively, we can write the following code.

my\_list = [char for char in 'hello']  
print(my\_list)

['h', 'e', 'l', 'l', 'o']

my\_list = [num for num in range(0,10)]  
print(my\_list)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Q. Square of numbers

my\_list = [num\*\*2 for num in range(0,10)]  
print(my\_list)

Q. Printing all even numbers.

#printing all even numbers  
my\_list = [num for num in range(0,10) if num%2 == 0]  
print(my\_list)

[0, 2, 4, 6, 8]

**Set comprehensions:**

my\_list = {num for num in range(0,10)}  
print(my\_list)

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

**Dictionary comprehensions:**

Important**:**

simple\_dict = {  
 'a' : 2,  
 'b' : 3  
}  
my\_dict = {key:value\*\*3 for key,value in simple\_dict.items()}  
print(my\_dict)

{'a': 8, 'b': 27}

simple\_dict = {  
 'a' : 2,  
 'b' : 3  
}  
my\_dict = {key:value\*\*3 for key,value in simple\_dict.items() if value%2 == 0}  
print(my\_dict)

{'a': 8}

Q. Find cube of numbers from one to ten as a dictionary.

my\_dict = {num:num\*\*3 for num in range(1,11)}  
print(my\_dict)

{1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216, 7: 343, 8: 512, 9: 729, 10: 1000}

Q. Check duplicate chars from a list.

some\_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']  
duplicates = []  
for i in some\_list:  
 if some\_list.count(i) > 1:  
 if i not in duplicates:  
 duplicates.append(i)  
  
print(duplicates)

Or by using comprehensions:

some\_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']  
  
duplicates = [x for x in some\_list if some\_list.count(x) > 1]  
print(duplicates)  
  
duplicates = set([x for x in some\_list if some\_list.count(x) > 1]) # sets do not hold duplicate  
print(duplicates)  
  
duplicates = list(set([x for x in some\_list if some\_list.count(x) > 1]))  
print(duplicates)

['b', 'b', 'n', 'n']

{'b', 'n'}

['b', 'n']

**Decorators:**

Have the @ sign and a name. e.g. @classmethod, @staticmethod, @route etc.

Decorators supercharge our function. So in below case, decorator gives some extra power to our function hello().

**@decorator**

**def hello();**

**pass**

Higher order function: A function is called Higher Order Function if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another function are known as Higher order Functions.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

*Decorator implementation:*

|  |  |
| --- | --- |
| def my\_decorator(func):  def wrap\_func(\*args, \*\*kwargs):  print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')  func(\*args, \*\*kwargs)  print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')  return wrap\_func  @my\_decorator def hello():  print("Hello, whats up!")  hello() | \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Hello, whats up!  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

Above, code can be understood in the following way:

hello2 = my\_decorator(hello) # hello function is pasased  
hello2()

i.e. wrapping the hello function with the decorator and assigning it to a varible.

Q. Create a decorator called performance to calculate the time taken by a function block.

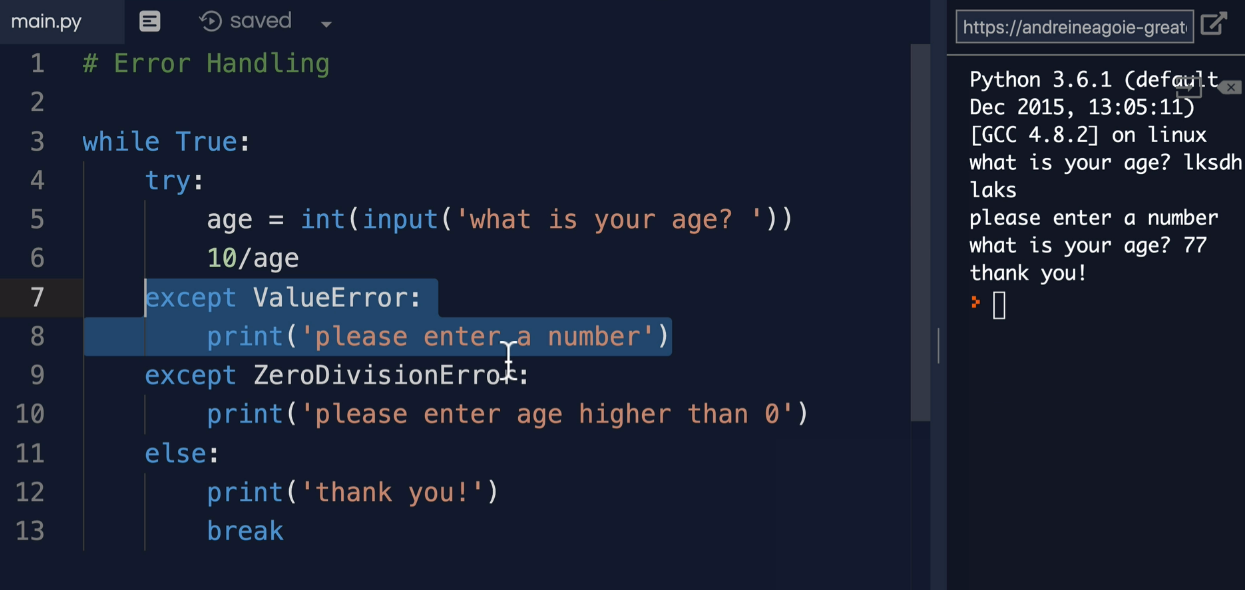
|  |  |
| --- | --- |
| from time import time   def performance(fn):  def wrapper(\*args, \*\*kwargs):  t1 = time()  result = fn(\*args, \*\*kwargs)  t2 = time()  print(f"It took {t2 - t1} sec to run.")  return wrapper   @performance def long\_time():  for i in range(10000000):  i \* 5   long\_time() | It took 0.2502412796020508 sec to run. |

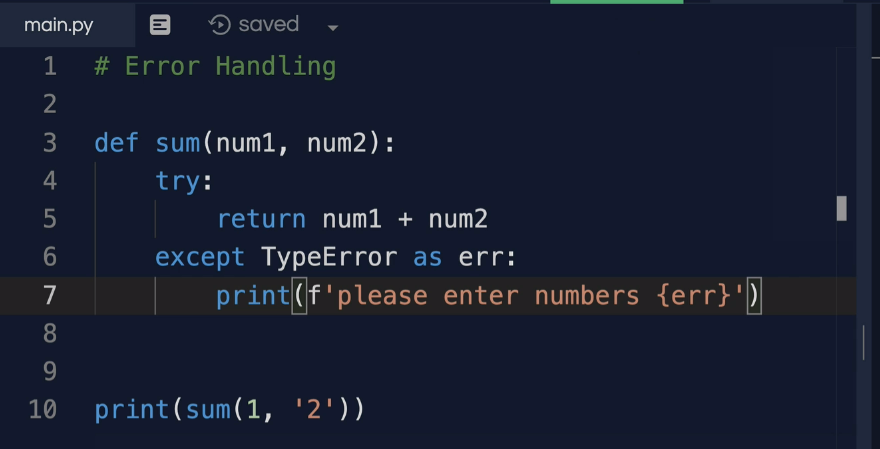
***Error Handling***

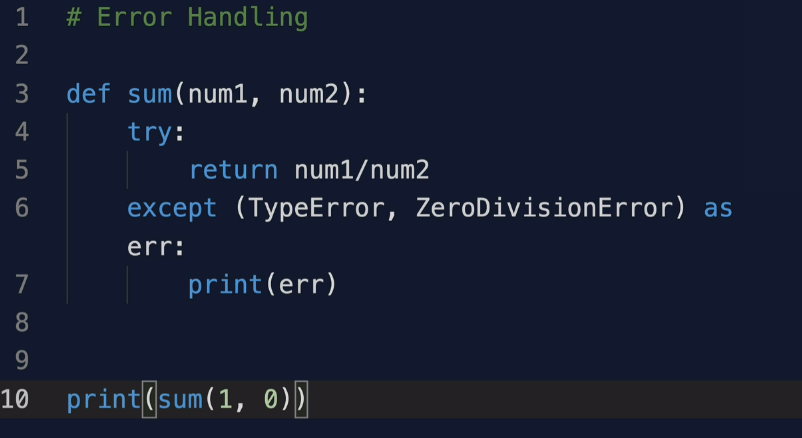
**try - except - else - finally**

|  |  |
| --- | --- |
| # Exception handling  while True:  try:  age = int(input("What is your age?"))  print(f'you are {age} year old...')  except:  print("Pls enter a number as age")  else:  print('Thank you.')  break | What is your age?>? asadf  Pls enter a number as age  What is your age?>? twenty  Pls enter a number as age  What is your age?>? 20  you are 20 year old...  Thank you. |

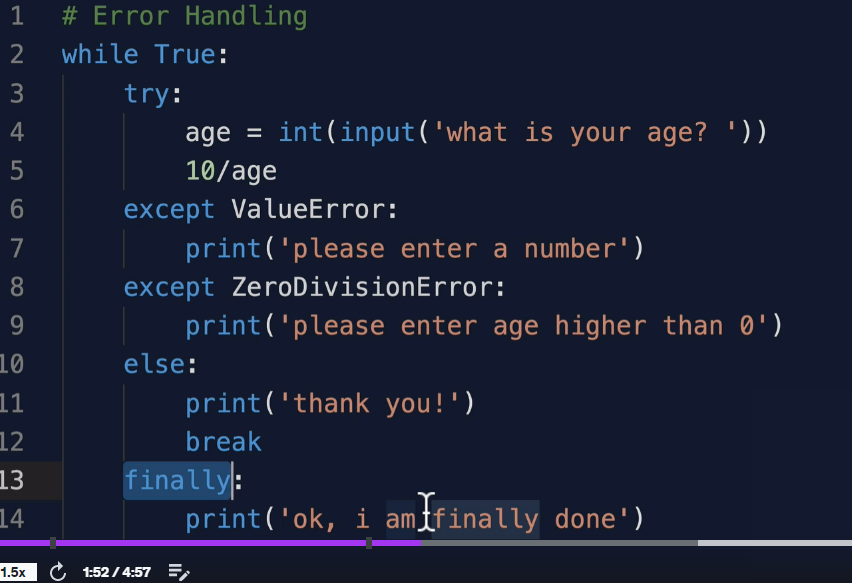
Some examples:







**finally** will run at the end, no matter what:



The **raise** keyword is used to raise an exception.

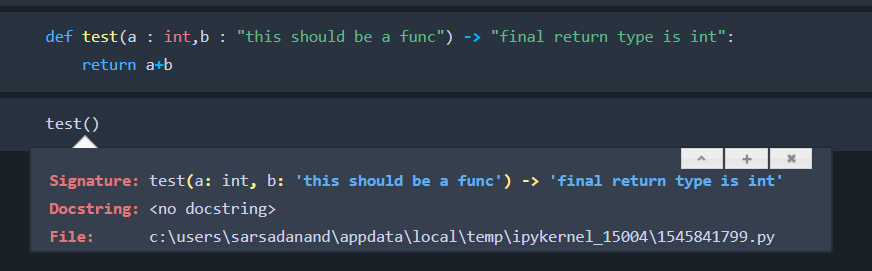
You can define what kind of error to raise, and the text to print to the user.

x = "hello"  
  
if type(x) is not int:  
 raise TypeError("Only integers are allowed")

# instead of TypeError, just Exception can also be given

TypeError: Only integers are allowed

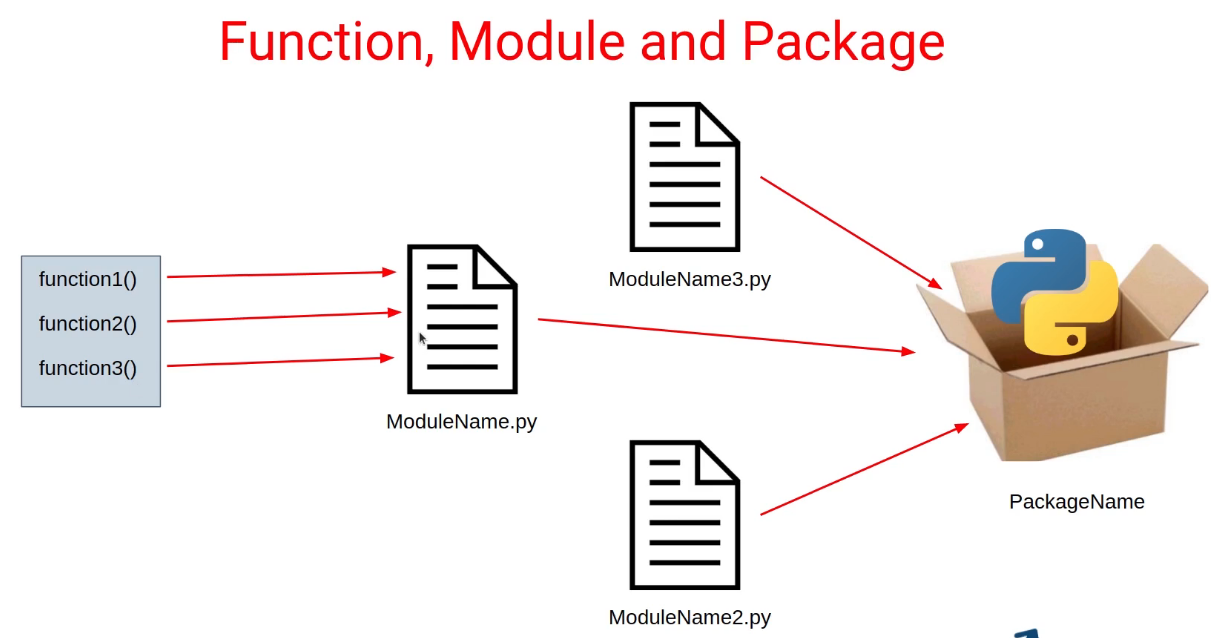
***Using hints in arguments while defining a function:***



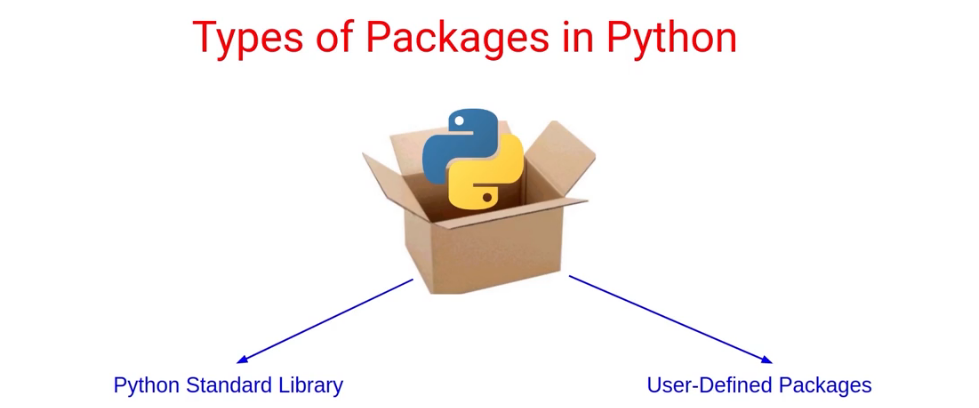
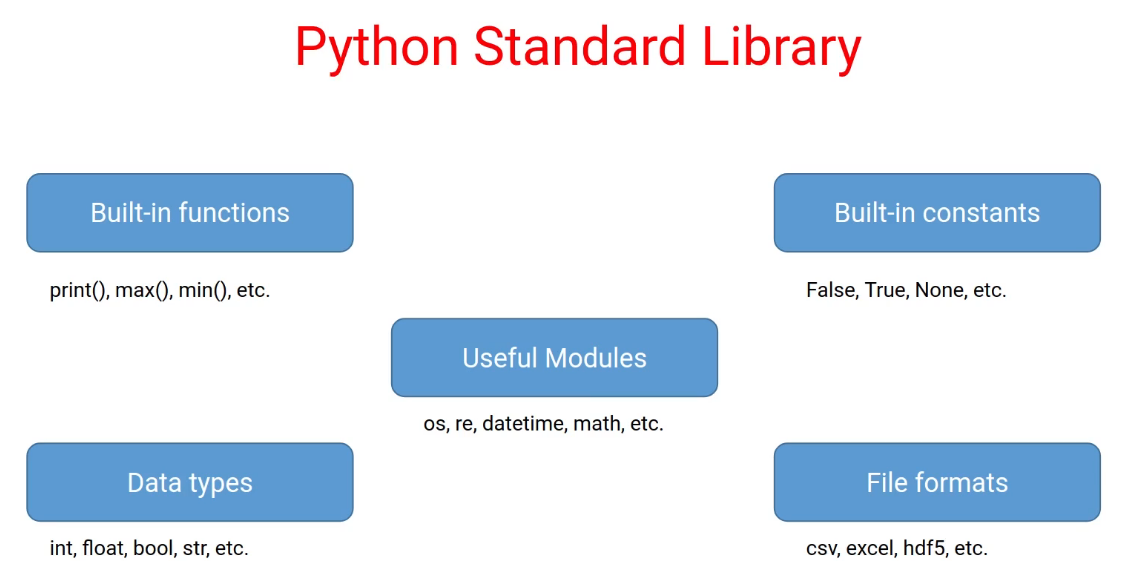
***Module and Packages:***

Module is simply a .py file which contains your function.

|  |  |
| --- | --- |
| **Circle.py***:*  def circle\_area(rad : float):  area = 3.14 \* rad \* rad  return area |  |
| *In the main python file/ipynm we will import the above module:*  import Circle  Circle.circle\_area(6) |  |



**A module is a collection of code or functions that uses the .** **py extension.** **A Python library is a set of related modules or packages bundled together**.

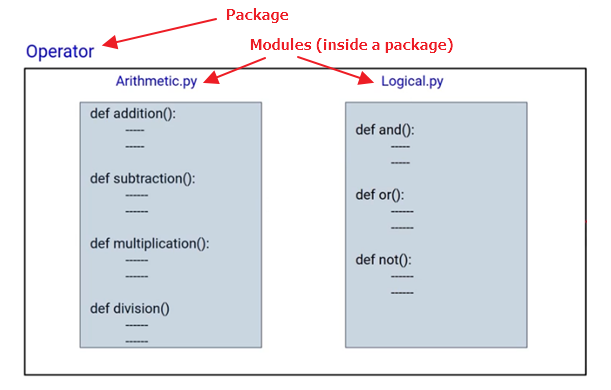
*** ***

Python Libraries

Actually, this term is often used interchangeably with “Python package” because packages can also contain modules and other packages (subpackages). However, it is often assumed that while a package is a collection of modules, a library is a collection of packages.

***Package:*** Is a collection of related modules. That aim to achieve a common goal.

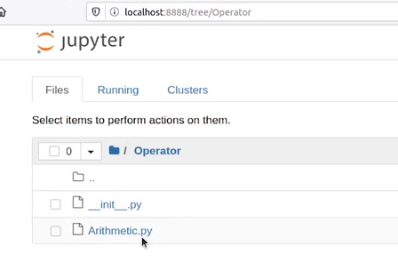
e.g. operator package below:



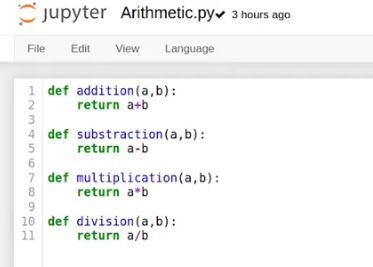
From above, Operator is the package and Arithmetic is the module which has 4 functions.

* from Operator import Arithmetic # to import entire module from a package
* from Operator.Arithmetic import addition # if you know the exact function you want to import from a package

To create a module, we need to have a \_\_init\_\_.py file along with the other modules (.py files).



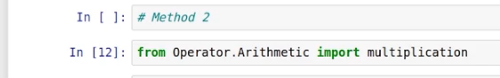
Note: In this case, it is an empty \_\_init\_\_.py file.



Now outside of Operator dir, create a ipynb and use this package.



To directly use the fuction in the code w/o using the name of module:

******

[***https://pypi.org/***](https://pypi.org/) To search/publish/download packages.

***Generators***