**Python**

IDLE is Pythons own IDE  
You can check type by type(variable)  
Variable type can change. Python is dynamically typed challenge.  
In Python 2, ½ would give 0 and in Python 3, result is 0.5  
Modulo operator – 5%3 – gives remainder which is 2  
  
Ordering in Python – BODMAS  
Brackets, Order(square,cube etc), Division, Mul, Add, Sub

round(1.5) – 2  
math.floor(1.5) – 1  
math.ceil(1.1) – 2  
You can do trigonometric functions like math.cos(number), math.pi(), find hypotenuse(3,4), other functions like math.exp, math.pow, math.log  
  
If a string is in next lines, use triple quotes (using single or double quotes)  
“””abc  
pqr”””  
  
Taking input inputtaken = input(‘please enter your input here’)  
  
String formatting – String is immutable  
1. a=’some string’, b=’other string’, you can do c = a+b  
2. a=’test’, a \* 3 -> testtesttest  
3. a=’test’, b=3, a+str(b) -> test3  
4. a=’my’, b=’name’, c=’pk’   
 d=‘{} {}: {}’.format(a,b,c) -> my name: pk  
 d=‘{2} {0}: {1}’.format(a,b,c) -> pk my: name  
 You can declare values or ask users for inputs like name, age  
 userstringmodel = ‘your name is {} and you are {} years old’  
 userstring = userstringmodel.format(name,age)  
5. a=”myname is pk and age is pk”, a.count(“pk”) -> 2  
6. a.lower(), a.upper(), a.capitalize() -> only capitalizes the first letter of the String  
 a.title() -> capitalizes beginning letter of every word  
7. a.islower(), a.isupper(), a.istitle(), a.isalpha()-> True is string contains only alphabets and should not have space, a.isdigit() -> only numbers, a.isalnum() ->True if only alphabets and numbers  
8. a=’my name is’  
 a.index(‘name’) -> 4. If item not found, will give error. Hence use find  
 a.find(‘name’) -> 4, a.find(‘nom’) -> -1  
9. a.strip(‘somecharacter’), a.lstrip(‘sc’), a.rstrip(‘sc’)  
10. len(a)-> number of characters in a  
11. a[3] -> n  
 a[3:4] -> na  
 a[start:end:step]

12. You can convert string to list by split()  
 a = “my name is pk”  
 b = a.split() -> [“my”,”name”,”is”,”pk”]  
You can join back and make a sentence by “ “.join(b)

3 ways to use a variable in string  
“my name is “+name  
f”my name is {name}”  
“my name is {}”.format(name)

3 ways you can reverse a string  
mystring = “pkhcl”  
1. mystring[::-1]  
2. for char in mystring:  
 revstring = char + revstring  
3. For char in mystring:  
 newlist = mystring.insert(0,char)  
 “”.join(newlist)

Lists  
List is mutable. You can change the value at particular indexes any time  
mylist = [1,2,3,4,5]

1. Check if exists – if 1 in mylist:
2. How to delete element 1 – mylist.remove(1) 🡪 removes only first occurrence
3. How to delete element by index element – del mylist[i]

You can also do del mylist[0:2]  
You can also do by mylist.pop(index). Difference is this returns the deleted element in the response. But when you do del, it happens silently

1. Add an element to list – mylist.append(element)
2. You can also add elements by mylist+[6,7]. But difference with append is, append changes original list. With +, you need to reassign to original list mylist = mylist+[6,7]
3. To append a value to specific index, you can use insert.   
   mylist.insert(2,6) -> [1,2,6,3,4,5]  
   you can also mylist.insert(2,[6,7])
4. Get length by len(mylist)
5. Enumerate  
   When you want to iterate through list, you can simply do  
   for book in books:  
   but if you want to use index as well like for i,book in books: it will not return expected result. Hence you need to use for I,book in enumerate(list)

Sets  
Looks list with braces. Sets cannot contain duplicates. Even if you declare and print you can see only unique values.  
myset = {1,2,3,4,5,1,2}  
print(myset) -> {1,2,3,4,5}   
  
Set is not stored in order in Python memory. Hence if you say myset[0] will give error  
myset.discard(3) -> to remove 3  
myset.clear() -> removes all elements  
myset.add(1) -> adds 1  
myset.update([1,2,3]) -> to add more than 1 element

Tuples  
Looks like list without braces. Tuples are immutable.

mytuple = 1,2,3,’A’,’B’

1. mytuple[0:2] -> 1,2,3
2. You can convert array to tuple by mytuple = tuple(mylist)
3. You can declare multiple variables on left and assign to tuple/list/string  
   a,b,c = [‘1’,’2’,’3’] or “123” or ‘1’,’2’,’3’  
   -> a = ‘1’ b=’2’ c=’3’

Dictionary

Key and value within braces {}  
students = {‘pk’:25,’hc’:21}

1. students[‘pk’] -> 25
2. You can edit by student[‘pk’] = 26 -> {‘pk’:26,’hc’:21}
3. You can get only keys by students.keys() -> dict\_key([‘pk’,’hc’])  
   To get it as list so that you can use it, convert it – list(students.keys())
4. You can get only values by students.values() -> dict\_values([25,21])  
   You need to convert to list here as well
5. You can get items similar to keys and values by students.items() which is again a dict  
    dict\_items([(‘pk’,25),(‘hc’,21)])
6. You can fetch value by   
   students[‘pk’] -> problem with this is, if ‘pk’ is not present, it will throw keyerror  
   students.get(‘pk’) – returns None

You can do   
 import random   
 and then random.choice()   
 or you can do from random import choice

Mylist = [‘pk’,’hc’]  
 choice(mylist) – will pick one randomly  
  
range(start,end) -> takes from start to end-1 -> [start,start+1,….end-1]  
range(n) -> 0 to n-1  
range(start,end,step)

List comprehension  
Creating lists using single line code using for, if etc  
even\_numbers = [x for x in range(1,20) if x%2==0]

1. w = [‘pk’,’hc’]  
   You can also have something like [[w.upper(), w.lower(), len(w)] for w in words]  
   Output will be [[‘PK’,’pk’,2],[‘HC’,’hc’,2]]

Functions  
There are only two scopes – global and local  
a variable inside a function cannot be used in another function if not declared outside function  
global variable is outside function.   
if you want to overwrite a global variable, you can’t do within a function. You need to be in global scope which is outside function.  
For example if you have a variable a = 10, if you want to declare this as 15 in a function and print value within the function, you will see 15 but outside the function, value doesn’t get changed. It will remain as 10.  
You can overwrite inside if if its in global scope and not within aa function.  
a=5  
def fn1():  
 a=10  
print(a) -> gives 5  
  
To overwrite, you need to declare a as global  
a=5  
def fn1():  
 global a  
 a=10  
print(a) -> gives 10  
  
Remember, you can overwrite lists and dictionary by changing just a part of it  
Ex: a = [1,2,3]  
 def fn1():  
 a = 5 //This will not overwrite  
 a[0] = 4 //this will work even though global is not used

Functions  
1. you can call a method by passing parameters in order or by passing with fieldname and values.  
def sum(x,y):  
sum(5,6)  
sum(x=5,y=6)  
2. Default values in method declaration should be done in the end  
def sum(w,x,y=5,z=10)

Packing and unpacking  
Unpacking happens while calling function. Packing happens in method definition  
1. Unpacking   
 When you add a \* followed by variable name, this unpacks variable and treats as individual elements. Ex: print(\*[1,2,3,4] -> 1 2 3 4  
 print(\*”pk”) -> “p” “k”  
 my\_dict = {‘name’:’pk’,’id’:5}  
 When calling a function, you can use fnname(\*\*mydict) -> this will go as fnname(name=pk,id=5). Simiarly def fnname(\*\*kwargs): for key,value in kwargs: you can use key and value  
  
2. Packing  
 When you add a \* in the function parameter, it combines all the parameters passed while calling the function.   
 ex: def sum(\*numbers):  
 for number in numbers:  
 total = total + number  
 return total  
 You can call function by sum(1,2) or sum(1,2,3,4,5)  
  
 If you send just tuples, its called args. In above case numbers is args.  
 If you send key and value, it’s called keywordarguments kwargs.  
 ex: def somefn(\*\*kwargs):  
 somevalue = kwargs[“somefield”]  
 you can also do for key,value in kwargs.items()  
 somefn(somefield=somevalue,someotherfield=someothervalue)  
  
Single asterisk is used for positional arguments and double asterisk for keywordarguments.

Classes  
class Someclass:  
 color = blue   
 wt = 5  
  
someobject1 = class()  
print(someobject1.color) -> blue  
someobject1.color = green -> this works but does not change value of parameter in class itself  
print(someobject1.color) -> green

class methods  
  
\_\_init\_\_method is not mandatory. Its constructor. Its used only when you need to initialie some attributes.   
  
class someclass:  
def somemethod():  
 def \_\_init\_\_(self,someparamifneeded): -> this \_\_init\_\_ is a constructor  
 self.color = blue  
 self.wt = 12  
 self.shape = circle  
 No need to put return inside constructor method. First param in constructor is self any name is fine.  
 def \_\_del\_\_(self): -> this is a destructor  
 print(‘object is deleted’)  
  
you can define other methods that use self  
def method1(self):  
 self.color = green  
  
you can write coin flip logic as below:  
class Coin:  
 def \_\_init\_\_(self, somearg):  
 self.name=mycoin  
 self.color=blue  
 self.wt=5  
 def flip(self):  
 mylist = [‘head’,’tail’]  
 self.face = random.choice(mylist)  
   
coin1 = Coin()  
coin.flip()  
coin.face -> it gives head or tail. Next time you want to do, you need to call coin.flip() again  
del coin1 -> this prints object is deleted

Abstraction  
Instead of creating class with same functions but different values and features, you can use a class as abstraction and use this class in other implementation classes.   
class myAbstractClass:  
  
You can also see Inheritance below where a class has access to methods in parent class.

|  |  |
| --- | --- |
| class myAbstractClass:  def \_\_init\_\_(self,name,id,gender):  self.name = name  self.id = id  self.gender = gender    def \_\_del\_\_(self):  print('object is deleted')  def \_\_str\_\_(self):  return “somethingbasedoncondn”    def findmygender(self):  if self.gender == 'M':  print('I am male')  else:  print('I am female')    class myclass1(myAbstractClass):  def \_\_init\_\_(self):  data = {"name":'pk',"id":'5',"gender":'M'}  super().\_\_init\_\_(\*\*data)  myc = myclass1()  myc.findmygender() -> I am male | class myAbstractClass:  def \_\_init\_\_(self,\*\*kwargs):  for key,value in kwargs.items():  setattr(self,key,value)    def \_\_del\_\_(self):  print('object is deleted')  def \_\_str\_\_(self):  return “somethingbasedoncondn”    def findmygender(self):  if self.gender == 'M':  print('I am male')  else:  print('I am female')    class myclass1(myAbstractClass):  def \_\_init\_\_(self):  data = {"name":'pk',"id":'5',"gender":'M'}  super().\_\_init\_\_(\*\*data)    myc = myclass1()  myc.findmygender() |

Polymorphism  
I will have a method in myclass1 with same name as the one in abstract class but with different behavior. This is Polymorphism. This is method overriding. You need to write findmygender inside myclass1  
def findmygender(self): printsomething  
  
Note: When you just print myc, you will see long string with memory info blabla. To avoid that, you can have \_\_str\_\_ function as mentioned above and the return of this gets printed for myc

Encapsulation  
When you declare a method in a class as static, method belongs to class but it does not access the instance of the class meaning it doesn’t use self. It can be called on the class or with the class instance but it does not modify the state of object or class. You can use static method when you do not require instance-specific data.

Code is below

You need to use decorator @staticmethod

|  |
| --- |
| class MathsOps:  def \_\_init\_\_(self):  print('just self')    @staticmethod  def add(a,b):  return a+b  c = MathsOps.add(5,6)  print(c) -> 11 |

Another way of encapsulation, which is done on variables is, in the class, instead of self.somevariable, use double underscore. Self.\_\_somevariable. This variable becomes private and cannot be accessed from outside class. But within class, this variable can be used.  
If you want it be private and access outside class, you need to define a getter method in class which return \_\_somevariable

Decorator  
This is to modify behavior of a function without modifying original function.  
You can create a decorator function and inside that a wrapper function.   
Decorator function takes argument which is the calling function itself. In wrapper function the arguments are calling functions arguments.   
You need to return wrapper from decorator function. However, everything happens inside wrapper. You can either perform some steps before and after calling func (argument you take in decorator) or you can perform validation of arguments that come to wrapper fn and then return the func such that after wrapper fn performs some tasks, control goes back and executes the original calling function.  
  
You can change the behavior of function or perform additional tasks of original function by adding a annotation of decorator function on top of calling function.

|  |  |
| --- | --- |
| def decorator(func):  def wrapper(a,b):  print('task 1')  func(a,b)  print('task 2')  return wrapper  @decorator  def add(x,y):  print('sum is ',x+y)  return x+y  add(5,10) | def decorator(func):  def wrapper(a,b):  print(a)  if not isinstance(a,(int,float)):  print('is not number')  else:  print('is a number')  return func(a,b)  return wrapper  @decorator  def add(x,y):  return x+y  c=add(5,10)  print('sum is ',c) |

You can also call without using annotation as below  
somevariable = decorator(add)  
somevariable(5,10)

Generators  
Generators are iterable variables which generates values lazily one at a time and only on demand, instead of storing all values in memory at once. Hence making it memory efficient.   
  
You can create a generator function or use generator expression  
  
Generator function is created by using the word yield. Generator remembers its state after yielding a value and remembers its execution context. When resumed, it continues from last yield. For example, if you write list comprehension, the entire value gets calculated and populated in the memory. In data intensive apps, this is inefficient. In Generators, each value is calculated on demand and returned till needed saving memory and time. Useful for large datasets and streaming data.  
First you write method to perform intended task. Instead of return use yield. Difference is yield sends one value at a time and also executes lines after usage of yield till end of function.   
Second use a variable and call the function and pass the parameter value.   
Third, iterate through the variable and print each element.

|  |
| --- |
| #count from 1 to 10  def count\_up\_to\_n(n):  count = 1  while count<=n:  yield count  count = count+1  cs = count\_up\_to\_n(5)  for j in cs:  print(j) |

Generator expression is written using brackets (). It looks like list comprehension but with brackets. You can see that n is not used inside expression. End value is defined already.  
This is 2 step. First, Use variable and Write generator expression. Second, iterate through the variable and print.

|  |
| --- |
| mygenerator = (i for i in range(1,6))  for k in mygenerator:  if k == 3:  break  print(k) |

PyTest  
You need to install pytest using pip install pytest.   
You just need to run command pytest and it picks files which begins with test\_\*.py or \_test.py, you can customize test discovery.  
  
 calc.py test\_calc.py

|  |  |  |  |
| --- | --- | --- | --- |
| Original\_Class  class Calculator:      def sum(self,a,b):          return a+b      def div(self,a,b):          if b==0:              raise ValueError("cannot divide by 0")          else:              return a/b | import pytest  from calc import Calculator  def test\_sum():      calcu = Calculator()      assert calcu.sum(3, 5) == 8  def test\_div():      calcu = Calculator()      with pytest.raises(ValueError,match='cannot divide by 0'): calcu.div(3,0) | import pytest  from calc import Calculator  @pytest.mark.parametrize("a,b,expected",[(2,3,5),(-1,1,0)])  def test\_add(a,b,expected):      calcu = Calculator()      assert calcu.sum(a, b) == expected  //using this you can execute multiple test inputs. No.of test cases increases with no. of data | import pytest  from calc import Calculator  @pytest.fixture  def calcul():      return Calculator()  def test\_add(calcul):      assert calcul.sum(2, 3) == 5 |

Mock dependency  
  
 calc.py test\_calc.py

|  |  |
| --- | --- |
| import requests  def fetch\_mul():      return 10  def multiply(number):      mul = fetch\_mul()      return number \* mul  def updateDB(data):      status = requests.get('url')      return 'success' | from unittest.mock import patch  import pytest  import calc  @patch("calc.fetch\_mul")  def test\_multiply(mock\_fetch\_mul):      mock\_fetch\_mul.return\_value = 5      assert calc.multiply(4) == 20      mock\_fetch\_mul.assert\_called\_once()  @patch("calc.requests.get")  def test\_updateDB(mock\_response):      mock\_response.return\_value = 'success'      assert calc.updateDB('abc') == 'success'      mock\_response.assert\_called\_once\_with('url') |

PIP

PIP – python install packages.  
Installs from Python Package index.  
  
When you run pip install packagename, pip searches for the package in Python Package Index and retrieves metadata. Metadata includes all the dependencies and compatibility. All the dependencies will be installed as well. Pip downloads all the files (.whl,.tar,.gz etc) and extracts the package and installs in Python env.   
  
Path in windows is python3.1\lib\site-packages  
In virtual env, it is projectname/venv/lib/python3.1/site-packages/  
  
You can run pip install libraryname==version  
You can put it in requirements.txt as below and run pip install -r requirements.txt  
 libraryname==version  
 libraryname==version  
You can also install from githuburl pip install githuburl  
You can also install from local path pip install ./pathtopackage  
  
You can install the libraries in the current directory by pip install -t .  
  
To lock versions for future use, pip freeze > requirements.txt  
  
If you do not specify the version by just running pip install libraryname, pip takes latest stable version and installs. You can check the version installed by pip show libraryname  
You can also install min or range by   
pip install libname>=1.2.3 pip install libname>=1.2.3,<3.0

You can customize pip behavior by using pip.conf or pip.ini file

To avoid dependency version clash, use virtual environment as below.  
python -m venv myenv -> this will create a folder myenv with Scripts and Lib  
you need to activate this by .myenv/Scripts/activate  
Then install your dependencies pip install numpy and run your program.   
Finally run command – deactivate and this will exit from virtual env.  
  
-m in python -m is to run module as a script.

Miscellaneous

* casefold() is used during case-insensitive comparison to convert a string to lowercase. This is better than lower(), since this handles special characters as well  
  if somestring1.casefold() == somestring2.casefold()
* If you want to continue code in next line, you need to use \ at the end of first line
* Below is the structure of module, package and library  
   library/  
   package1  
   module1.py  
   module2.py  
   \_\_init\_\_.py  
   package2  
   module1.py  
   \_\_init\_\_.py  
   somemoduleoutsidepackage.py  
   main.py  
    
   \_\_init.py\_\_ was mandatory in directory to define something as package in python 3.3 or below. Afte this, its not mandatory. But ppl follow this.  
  It contains code to initialize a package. \_\_ denotes a special use and not related to scope. You can have code like below in \_\_init\_\_.py  
   package\_version = “1.0” -> you can declare version  
   from .module1 import function1 -> this is done so that another module which imports this package can access some info easily without having to go to specific modules since these are already imported in init file.   
   \_\_all\_\_ = [‘module1’,’module2’] -> if someone does import \*, only these modules will be allowed to be imported.   
    
  Accessing the package to demo init usage:  
   somepythonfile:  
   from package1 import function1, package\_version -> since these info are present in init already, didn’t have to go to specific module  
     
   main.py is a script outside library
* Global, protected, private  
  Global you know.. defined in global scope. To use inside fn, global keyword is needed  
  Protected vars are marked with \_ they can be accessed outside class but developers shouldn’t use. This is to notify that  
  Private vars are marked with \_\_ cannot be accessed outside the class. If you use, you will get AttributeError
* self is not a keyword. You can write anything. Inside \_\_init\_\_(whateveryouwrite) becomes self. It is used to represent instance of the class.
* Memory management in Python  
  Python has Stack memory and Heap memory.   
  If you declare somevar1 = 5, somevar1 holds the reference to the value. Not the value itself. This will have a memory address in stack. The value 5 is stored in another space in Heap memory. Memory address in stack will be mapped to value 5.   
  If you declare another variable somevar2 = 5, since somevar2 also has some value, it will have same memory address as somevar1. You can check this by id(somevar).  
  Now lets say, you delete the line somevar2=5, somevar2 will be removed from stack. But the value 5 will be stored in heap since it is referenced by somevar1.   
  If you now delete somevar1=5 or assign another value to somevar1=6, 6 will be stored in another location in heap and somevar1 will get another memory ID in stack. 5 will be removed from heap memory by garbage collector.   
    
  For mutable types like list, even if you alter some elements, memoryID of variable will not change. But for immutable types, it will change.  
    
  Remember, everytime you execute program to check memory ID, new IDs will be allocated. But just that, variables with same value will share same ID in stack.
* Multi-threading :   
    
  Global Interpreter lock ensures that only one thread executes Python bytecode at a time. Even if you use multithreading, only one thread can execute Python code at a time. While other wait for GIL to be released.   
  By default, Python uses single main thread. You can put some tasks to use a different thread by creating a new thread by threading.Thread(target=somemethodname) and starting it using start() method. Multithreading is good for i/o operations and not CPU bound operations.  
  If you want main thread to wait for these threads, you need to use join() method.  
  Example: main thread creates a thread for method 1 and 2 and then goes to execute print both threads are done. But since join is used, will execute print after both threads are actually completed.

|  |
| --- |
| Import threading  def method1():  for i in range(5):  print(‘a’) def method2():  for I in range(5):  print(‘b’)  thread1 = threading.Thread(target=method1) thread2 = threading.Thread(target=method2)  thread1.start() thread2.start()  thread1.join() thread1.join()  print(‘both threads are done’) |

* You can remove a file by os.remove(filename)
* Shallow copy and deep copy:  
  Shallow copy is when you copy an object all the nested values will not be copied in the new object. Instead reference will be made to old object. If you edit something in old object, new object also gets changed.   
  Deepcopy: This copies a old object along with all nested values and creates new object. This way even if you edit old object, new object does not get affected.
* Zip  
  Zip is used to combine two iterable items like list, tuple and retains the shortest length discarding other items.   
    
  a = [ ‘pk’,’hc’]  
  b = [1,2,3]  
  c = zip(a,b) -> [(‘pk’,1),(‘hc’,2)]
* Some built in modules – sys, os, math, random, datetime, json
* Xrange and range  
  xrange would return a generator object and we had to iterate through this till the required point. From Python 3, xrange is not supported.
* Some run time exists in Python ex: missing parenthesis in print statement

CICD  
Gitlab CICD steps  
Gitrepo – Tests – Build&Package – ArtifactRepo – Deployment.  
  
Advantage of Gitlab CICD is it already has our code. Seamless integration to repo.   
In Jenkins, you need to setup server. In Gitlab, you don’t need to.   
  
Gitlab runner is an agent to run CICD jobs. Pipeline assigns jobs to available runners. You can use shared runner or create one of your own on any server like ec2  
  
You can have makefile. You can declare steps like run, test, build etc in makefile.  
You can call make run, make test etc. These commands will go under script in jobs in gitlab ci. You can do make run to run the application locally. Something like venv path/activate and python app.py  
  
Gitlab runners execute on Docker container. You can define the type of container you want by choosing the container image. By default, it uses Ruby.   
For Python, you need Python, pip, make. For this, you need to use Python image. Better to define the specific version of image instead of putting latest. You can also use ECR here.   
You can also use another repository to build image with necessary apps and create an image in ECR.  
  
In Gitlab CI file, you need to install make before you execute script like make test.  
Hence make needs to be installed in before\_script as apt-get install make.   
  
To use secrets in Gitlab, you need to create variables under variables section on Gitlab UI. How this is secure is, only people with privileged gitlab access can see this section on gitlab UI.   
  
In Gitlab CI file, you will have 3 steps.   
1. Test – install make in before\_script and in script, execute make test  
2. Build – steps like docker login, build, push  
3. Deploy – docker login, ps, run on a port  
  
In Dockerfile, you can see steps like   
pip install -r requirements.txt  
uvicorn command as wellS  
  
<https://gitlab.com/nanuchi/gitlab-cicd-crash-course/-/tree/main/>