**PARTE 2 DE LEGUAJE DE PYTHON**

descomposicion del proyecto

es cuando se divide el proyecto en partes mas pequenas

-interfaz de usuario

-logica del proyecto

en programacion las divisones resultantes se llaman **modulos**

usuario : si usas el modulo creado por otra persona

proveedor: cuando tu creas el modulo para ti o para otros

cada modulo consta de entidades

son funciones, variables, constantes, clases y objetos

**IMPORTAR MODULOS** (METODO 1)

import nombre\_modulo1, nombre\_modulo2

import nombre\_modulo1

import nombre\_modulo2

esta instrucción se puede colocar en cualquier parte del programa **pero** debe estar antes **del primer uso de las entidades del modulo**

**se importan TODAS las entidades del modulo**

namespace es el “espacio” donde exite el nombre de la variable o funcion

se puede tener en nuestro propio programa variables ofunciones esactamente con el mismo nombre que las del modulo importado

USO DE ENTIDADES DE UN MODULO

nombre\_modulo.variable

tenemos que poner el nombre del modulo despues un punto (.) y despues el nombre de la entidad , variable o funcion

**ALIASING** (RENOMBRAR MODULOS)

**solo en el metodo 1** se puede cambiar el nombre del modulo, para facilitar su uso con las entidades

import modulo as alias

as : hace que el modulo tome el nombre del alias

**IMPORTAR MODULOS** (METODO 2)

from modulo import entidad1,entidad2

from math import sin,pi

print(sin(pi/2))

se pone el nombre del modulo a importar y las entidades espesificas que se desea importar de ese modulo

para usar las entidades solo es necesario poenr el nombre propio de la funcion

**solo se importan las entidades especificadas en el comando**

la importacion se la puede hacer en cualquier parte del codigo

al momento de importar las entidades especificas del modulo con este metodo

las variables que teniamos antes de la importacion cambian de valor o funcion

a las del modulo importado

si declaramos entidades con el mismo nombre que las del modulo despues de importar el modulo

los valores de estas entidades cambian a las nuestras y olvida las del modulo

**ALIASING (**CAMBIAR EL NOMBRE DE LA ENTIDAD)

**en el metodo 2** se puede cambiar el nombre de las entidades

from modulo import entidad1 as alias1 , entidad2 as alias2

cambiamos el nombre de las entidades para usarlos despues

**DIR(MODULO)**

**nos devuelve** una lista con **todas** las entidades del modulo

es necesario importar el modulo con el metodo 1 antes de usar este comando

si el modulo tiene un alias se usa este comando con el alias

**IMPORTAR MODULOS** (METODO 3)

from modulo import \*

esta es una convinacion de los dos metodos anteriores

importa todas las entidades del modulo

no es necesario poner modulo.entidad para usar la entidad1

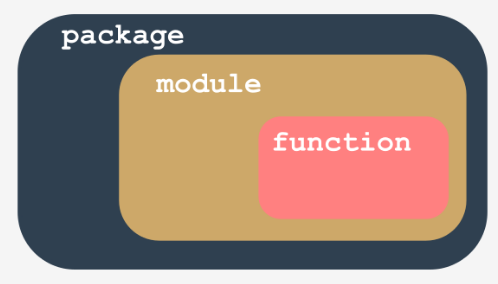
solo se pone el nombre de la entidad

al no conocer todas las entidades del modulo puede haber conflicto o hacer varial las entidades de nuestro programa

**ESCRIBIR MODULOS**

un modulo es una agrupacion de funciones

un paquete es una agrupacion de modulos

al momento de hacer de ejecutar nuestro primer import de nuestro modulo aparecera un directorio en el mismo lugar donde creamos nuestro modulo

llamado \_\_pycache\_\_.

Dentro de este directorio se crea un archivo llamado

nombre\_modulo.implmentacion\_python-version\_python,pyc

modulo.cpython-36.pyc

pyc significa python y compilado

en el caso de los paquetes este directorio se crea al nivel donde haya modulos para ser importados

este directorio solo se crea cuando importamos por primera vez el modulo

dentro del directorio

este archivo .pyc es una forma “semi compilada” del codigo que escribimos

es un archivo listo para ejecucion

su ejecucion es mas rapido

la actualizacion de este archivo en caso de modificar el .py

cuando se importa un modulo este se ejecuta implicitamente

si se importa varias veces el mismo modulo python solo hara caso a la primera vez

en caso de asignarse variable por el modulo solo se las hara la primera vez

en el modulo se crea una variable:

\_\_name\_\_

si la ejecutamos mediante el modulo no dira que la variable tiene

\_\_main\_\_

si importamos el modulo nos dara que la variable tiene el nombre del modulo

\_\_name\_\_ = modulo

es un **convenio no regla** poner guion bajo (₎ o guion bajo doble (\_₎ para las variables que no se deben cambiar

pero los usuarios pueden o no hacer caso, no hay nada que lo impida

#!/usr/bin/env python3

este es un shebang line

especifica que este archivo se debe correr con python3

#!/usr/bin/python3

se puede usar este pero el de arriba es mas portable

**MODULO: SYS**

form sys import path

path.append(‘/home/rodrigo/Escritorio/curso\_python/programas\_2

ponemos estas dos lineas para que sepa donde buscar los modulos

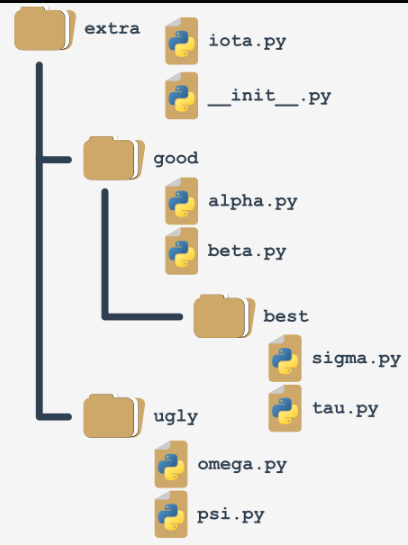
la variable path del modulo sys esta guardado las rutas de donde buscar los modulos cuando se los invoca

**PAQUETES(**MODULOS DE MODULOS)

en el **directorio raiz** tenemos que poner un fichero llamado

\_\_init\_\_.py

este fichero puede estar vacio o tener la inicioalizacion de todos demas modulos pero siempre tiene que estar en la raiz del paquete



**MODULOS**

MODULO: math

sin(valor) , cos(valor) ,tan(valor) : con las funciones trigonometricas

valor debe estar en **radianes**

asin(valor) , acos(valor) , atan(valor) : son las funciones arco

toman un valor y el resultado es en **radianes**

pi : es el valor pi

e : constate de euler

radians(valor) : esta funcion convierte de **grados a radianes**

valor tiene que estar en grados

degrees(valor) : esta funcion convierte de **radianes a grados**

valor tiene que estar en radianes

exp(valor) : es la constante de euler elevado a la potencia del valor

log (valor) : es el logaritmo natural de valor

log(valor , base) : es el logaritmo en base (especificado) de valor

log10(valor) : es lo mismo que log(valor , 10)

losg2(valor) : es lo mismo que log(valor , 2)

pow(base , exponente) : es la potencia de la base elevado al exponente especificado

esta funcion ya esta dentro de python

ceil(valor) : **devuelve** el entero inmediato superior de valor

floor(valor) : **devuelve**el entero inmediato menor de valor

trunc(valor) : **devuelve** el valor truncado

factorial(valor) : **devuelve** el factorial de valor

hypot( x , y ) : **devuelve** el valor de una hipotenusa con catetos ‘x’ y ‘y’

MODULO : random

random() : esta funcion **devuelve** un valor **flotante** pseudo aleatorio

el valor esta entre 0.0 y 1.0

seed(a=None , version =2) : establece la semilla determinara en que orden apareceran los valor **pseudoaleatorios**

el argumento version dice si es 2 usa el algoritmo de python3 y si es 1 usa el de python2

seed() : dejarlo en blanco usa la hora dela compu como semilla

seed(valor\_int) : establece el valor entero como semilla

si establecemos una semilla manualmente, los valores generados en cada ejecucion del programa seran **siempre** los mismos

randrange(fin) : **devuelve** valores **enteros** que van desde **0 hasta un numero antes que fin**

va desde 0 hasta fin-1

randrange(inicio,fin) : genera enteros que **van desde inicio hasta fin-1**

randint(inicio,fin) : **devuelve** enteros que van **desde incio hasta fin incluido**

choice(lista) : esta fucnion **devuelve** un valor aleatorio escogido de la lista dada

las lista puede contener lo que sea

choice(range(valor)) : se lo puede usar asi

sample(lista,numero\_extraido) : **devuelve** una **muestra de numero\_extrido valores** aleatoria de la lista

numero\_extraido tiene que ser menor que el tamano de la lista

sample(range(valor), numero\_extraido) : se lo puede usar asi

MODULO : platform

permite acceder al subyasente

al hardware, sistema operativo e informaciond sobre la version del interprete

platform(aliasing=0,terse=0) : nos **devuelve** informacion del plataforma subyacente (sistema operativo)

platform() : nos da toda la informacion que pueda

platform(1) : nos presenta los alias, para que sea mas entendible

platform(0,1) : nos da la informacion minima

machine() : nos **devuelve**  el tipo de procesardor que usa la maquina

system() : devuelve el nombre generido del sistema operativo

version() : devuelve la version del sistema operativo

ubuntu,debian,kali

python\_implementation() : nos devuelve la **implementacion** de python que se esta usando

cpython, jpython, PyPy ...etc

python\_version\_tuple() : nos **devuelve una tupla** con la version de python qu estamos usando

(‘3’,’6’,’9’) → python3,6,9

**EXCEPCIONES (**ERRORES)

try:

comandos a realizar normalmente()

except nombre\_error1:

comandos que se ejecutan en caso de que la excepcion 1 ocurra()

except nombre\_error2:

comandos que se ejecutan en caso de que la escepcion 2 ocurra()

except (nombre1 , nombre2):

comandos que se ejecuten cuando **pasan nombre1 O nombre2**

else:

comandos a ejecutarse en caso de que no salte ninguna excepcion

dentro de try se pone nuestro codigo que queremos ejecutar

en caso que haya un error el progrma saltara a except

aquí es odnde tenemos que tratar de resolver el los errores

despues pasa alas lineas despues del bloque except

except:

poner un except sin ningun nombre, es como el default

los errores que no se identifiquen iran directo a este

se lo tiene que poner al final de todos los excetp

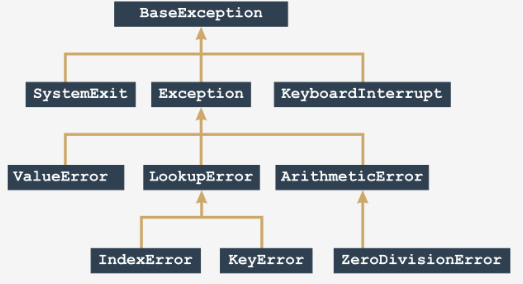
el nombre de la excepcion es un objeto de clase de la excepcion

ver ejemplo en /programas\_2/codigo10.py

print(nombre\_excepcion.\_\_str\_₍))

esto nos dara un mensaje mas detallado de lo que trata la excepcion

NOMBRE DE ERRORES



BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

+-- StopIteration

+-- StandardError

| +-- BufferError

| +-- ArithmeticError

| | +-- FloatingPointError

| | +-- OverflowError

| | +-- ZeroDivisionError

| +-- AssertionError

| +-- AttributeError

| +-- EnvironmentError

| | +-- IOError

| | +-- OSError

| | +-- WindowsError (Windows)

| | +-- VMSError (VMS)

| +-- EOFError

| +-- ImportError

| +-- LookupError

| | +-- IndexError

| | +-- KeyError

| +-- MemoryError

| +-- NameError

| | +-- UnboundLocalError

| +-- ReferenceError

| +-- RuntimeError

| | +-- NotImplementedError

| +-- SyntaxError

| | +-- IndentationError

| | +-- TabError

| +-- SystemError

| +-- TypeError

| +-- ValueError

| +-- UnicodeError

| +-- UnicodeDecodeError

| +-- UnicodeEncodeError

| +-- UnicodeTranslateError

+-- Warning

+-- DeprecationWarning

+-- PendingDeprecationWarning

+-- RuntimeWarning

+-- SyntaxWarning

+-- UserWarning

+-- FutureWarning

+-- ImportWarning

+-- UnicodeWarning

+-- BytesWarning

**ver pagina** <https://docs.python.org/2/library/exceptions.html>

para ver los detalles de cada erro

la gerarquia de excepciones: mientras mas arriba este es mas general y abstracta

mientras mas abajo son mas especificos

una rama inferior se la puede generalizar con una rama superior

la excepcion que identifique el error y aparesca primero en el programa lo atrapara

poner las excepciones generales al final, poq sino las especificas nunca actuaran y seran inutiles

la excepcion llamada “exception” se le puede pasar argumentos para

GENERAR EXCEPCIONES

raise nombre\_excepcion

este comadno simula la excepcion que se le especifique

assert expresion\_condicion , ‘mensaje ‘

assert evalua la exprecion de ocndicion

en caso de **ser true NO hace nada**

en caso de **ser false lanza una excepcion** llamada:

AssertionError

ademas muestra el mensaje que le pasemos

CREAR EXCEPCIONES

ya que las excepciones predefinidas son nada mas que clases, se puede crear nuevas partiendo de la herencia de alguna de las ramas predefinidas

class miExcepcion(Exception):

pass

estamos creando la excepcion heredando la clase exception, entonces estamos agregando una nueva rama al arbol

**ver ejemplo en /**programas\_2/codigo10.py/

**se puede crear una nueva rama de escepciones**

**CODIGO ASCII**

utiliza 8 bits para la representacion de cada carácter

solo hay 256 caracteres

en linux para meter ascii es

ctrl + shift + u + **el codigo ascii en HEXADECIMAL**

PUNTO DE CODIGO

es el valor decimal que representa los caracteres

PAGINA DE CODIGO

como los 128 primero codigos esta asignado para los caracteres “latinos”

los siguientes 128 codigos se los usa para representarlos dependiendo para que ligar se lo usa (europa, occidente, alfabeto griego, arabe o hebreo)

cada pagina de codigo es distinto según para donde este aseignado

los estandares ISO/IEC 8859-X especifica los caracteres y para donde esta asignado

ISO/IEC 8859-2 para lenguas esclavas

ISO/IEC 8859-5 para lentras cirilicas

**UNICODE**

asigna caracteres a mas de un millon de puntos de codigo

UTF-8

8 bits para caracteres latinos y los ASCII

16 bits para los carateres NO latinos

24 bits para ideografos CJK (china-japon-corea)

python3 trabaja con UNICODE y UTF-8

COMANDOS RELACIONADOS

ord(caracter) : esta funcion nos devuelve el numero decimal de ascii del caracter que se le pase

chr(numero\_ascii) : esta funcion devuelve el carácter ascii del numero decimal que se le pase

min(cadena) : nos devuelve el carácter con el valor ascii mas bajo

max(cadena) : nos devuelve el caracter con el valor ascii mas alto

cadena.index(caracter) : devuelve la posicion de indice de la ubicación del carácter en la cadena

devuelve el indice de la primera coincidencia

**s**i no encuentra coincidencia da error

cadena.find(caracter , indice\_inicio, indice\_final) : funciona igual que .index() pero en caso de no encontrar coincidencia devuelve -1

el indice\_inicio dice a partir de que indice empezara a buscar

si hay coincidencia **antes**  de indice\_inicio, ignorara estos

termina de buscar hasta limite\_final

cadena.rfind(caracter , indice\_inicio , indice\_final)

funciona igual que .find(), pero empieza a buscar de derecha a izquierda

cadena.count(caracter) : devuelve el nemero de veces que aparece el carácter en la cadena

cadena.capitalize() : devuelve la misma cadena pero con el primer carácter en mayuscula y el resto en minuscula

en caso de que el primer carácter no sea una letra no solo vuelve a minusculas el resto de **letras**

cadena.center(numero\_caracteres , ‘\*’) : devuelve la cadena centrada dentro del numero de caracteres especificados,

el segundo argumento dice que carácter pondra a los espacios restantes

print(‘[‘ + ‘hola’.center(6,’\*’) + ’]’)

[\*hola\*]

si numero\_caracteres es mas pequeño que el tamano de la cadena devuelve la cadena original

cadena.endswitch(subcadena) : devuelve True si la cadena **termina** con lo que contine la subcadena

cadena.startswitch(subcadena) : devuelve True si la cadena **empieza** con lo que contiene subcadena

cadena.isalnum() : devuelve True si la cadena **esta formada solo por letras y numero**

devuelve False si hay otro carácter (.,\*@)

cadena.isalpha() : devuelve True si la cadena  **solo tiene letras**

si hay otra cosa devuelve False

cadena.islower() : devulve True si la cadena **tiene solo LETRAS MINUSCULAS**

cadena.isupper() : devuelve True si la cadena **tiene solo LETRAS MAYUSCULAS**

cadena.isdigit() : devulve True si **en la cadena solo hay digitos**

devuelve False si hay otra cosa

cadena\_separador.join([lista\_de\_cadenas]) : el metodo **devuelve una cadena**

la nueva cadena contiene las cadenas de la lista separados por cadena\_separador entre ellos

cadena.split() : **devulve una lista** con las subcandenas separadas por espacio de la cadena

cadena.lower() : **devulve** la cadena con todas sus letras **en minusculas**

cadena.replace(subcadena\_original , subcadena\_nueva , cuantas\_veces)

devuelve la candena pera **reemplaza todas** las coincidencias de subcadena\_original por subcadena\_nueva

si se omite el tercer argumento reemplazara todas las coincidencias

el tercer argumento dice **cuantas coincidencias reemplazara**

el resto de coincidencias las mantendrá como el original

cadena.swapcase() : devuelve una nueva cadena donde cambia las minusculas a mayusculas y viceversa de la cadena original

cadena.title() : devuelve una nueva cadena con **la primera letra de CADA PALABRA** en mayuscula y el resto a minuscula

**MULTILINEAS**

cadena= ‘’’linea #1

linea #2’’’

la cadena toma todo el texto como la cadena, tomando en cuenta el ‘\n’ como salto de linea

**PROGRAMACION ORIENTADA A OBJETOS**

los metodos son datos

son funciones que se invocan desde dentro de los datos

los objetos :

tiene nombre: es unico, pero puede ser anonimo

tienen atributos o propiedades (sinonimos aquí)

pueden realizar un conjunto de actividades que se llaman metodos

la clase:

es un conjunto de objetos

**comentario :** cuando creamos una una clase en python **siempre** esta clase hereda una superclase llamada objeto

CLASES

class nombre\_clase:

pass

print(f’variables de clase : {nombre\_clase.\_\_dict\_\_}’)

imprimira las **variables de clase y las funciones dela clase**

CREAR OBJETOS

nombre\_objeto1 = nombre\_clase()

nombre\_objeto2 = nombre\_clase()

nombre\_objeto3 = nombre\_clase()

el objeto esta equipado con todas las propiedades de la clase

esto se llama instanciacion

el objeto se convierte en una instancia de la clase

cuando se crea el objeto estos **adquieren automaticamente** un conjunto de propiedades y metodos

nombre\_objeto.\_\_dict\_\_

\_\_dict\_\_ es un diccionario el cual **posee todas la variables** del objeto en el momento

si la variable es privada \_\_dict\_\_ postrara asi:

\_nombreClase\_\_variable

solo si accedemos a la variable privada de esta manera se puede acceder a ella, de otra manera no se puede acceder a una variable privada

print(\_nombreClase\_\_variable)

y las variables nuevas

print(f’variables de la clase : {nombre\_objeto.\_\_dict\_\_}’)

nos saldra solo las variables de de instancia

JUNTAR TODO

class pila: #definimos la clase pila

def \_\_init\_\_(self): #definimos la funcion contructor

self.listapila=[] #le estamos agregando una propiedad al objeto

#self.\_\_listapila=[] #declaramos la propiedad privada

def \_\_str\_\_(self):

funciones o comandos

return cadena

objetopila = pila() #instanciamos el objeto

ver codigo7.py de programacion

para acceder a las propiedades de los objetos se usa punto(.) como con los metodos, pero no se pone parentesis al final

cuando se pone dos guiones bajos (\_\_listapila₎ antes del nombre de la variable la declaramos como privada, y la variable solo se puede acceder desde la clase

self.\_\_milista = []

esto se llama encapsulacion

significa que no se puede alterar la variable fuera de ella clase

el metodo \_\_str\_\_ es un metodo que se ejecuta automaticamente al instanciar la clase, como \_\_init\_\_

pero este metodo si puede retornar cosas y realizar funciones

ATRIBUTOS PREDEFINIDOS DE LAS CLASES

type(nombreObjeto)

esto nos dara la clase del objeto

nombreClase.\_\_name\_\_

contiene una cadena, con el nombre de la clase nada mas

esta variable solo exite para las clases,**no para los objetos**

nombreClase.\_\_module\_\_

contiene el nombre del **modulo** (archivo/fichero)

si devuelve \_\_main\_\_

significa que el fichero (nombreClase) se esta ejecutando, osea que es el fichero sobre el cual estamos

nombreClase.\_\_bases\_\_

contiene una tupla que **contiene clases** (no nombres de clases) **que son superclases directas de la clase actual**

esta variable solo se puede usar con clases, **no con objetos**

si devuelve ‘object’

significa que la la clase especificada no tiene superclases explicitas

hasattr(nombre\_objeto , ‘propiedad’)

hasattr(nombre\_clase , ‘propiedad’)

este comando pregunta si la clase o el objeto posee la propiedad especidicada

devuelve True si clase poesee el objeto

propiedad :puede ser variable,funcion o metodo

la propiedad siempre se le tiene que mandar como cadena

entre comillas

issubclass(nombreClase1 , nombreClase2)

esta funcion devuelve True **clase2 es superClase de clase1**

si no lo es devuelve False

getattr(nombreObjeto , ‘nombrePropiedad')

devuelve el valor actual de la propiedad pertenecietne al objeto

nombrePropiedad es solo el nombre de la variable **en forma de cadena**

isinstance(nombreObjeto , int)

setattr(nombreObjeto , nombrePropiedad , nuevoValor)

le asigna nuevoValor a la propiedad (nombrePropiedad) del objeto (nombreObjeto)

nombrePropiedad tiene que ser una cadena

METODOS Y FUNCIONES

cuando el nombre de un metodo empieza con dos guiones bajos (\_\_) es poq estos estan ocultos

como el caso de \_\_init\_\_

a estos metodos se puede acceder como con las variables ocultas

class miclase:

def \_\_oculto(self):

print(‘esoty oculto’)

objeto = miclase()

objeto.\_miclase\_\_oculto()

constructor: \_\_init\_\_

este no es un metodo comun, **es un metodo constructor**

esta clase se invoca automáticamente cuando se instancia un objeto de la clase

se puede usar este metodo para inicializar variables de instancia,para isntanciar otros objetos

el constructor no puede retornar ningun valor

lo que si retorna es **un objeto**

SELF

poner ‘self’ en todas las funciones de la clase es necesrio

asi el metodo accede a todas las propiedades, actividades/metodos de la clase

para acceder y crear a los atributos **DENTRO LA** clase se tiene que hacer a travez de ‘self.’

self.milista = []

self.milista.append(valor)

del self.milista[0]

self identifica a cada objeto para el cual se invoca el metodo

class conClase():

def otro(self):

print("otro")

def metodo(self):

print("método")

conClase.otro()

obj = conClase()

obj.metodo()

self tambien sirve para invocar otros metodos de la misma clase

VARIABLES DE INSTANCIA

cuando creamos un objeto y le asignamos valores a sus variables estas se llaman variable sde instancia

una vez creado el objeto se le puede agregar propiedades (variables) apartes de los de la clase del objeto

nombre\_objeto.nueva\_variable = 0

VARIABLES DE CLASE

se la inicializa dentro de la clase pero fuera de cualqueir metodo de la misma

estas variables no se muestra en el objeto \_\_dict\_\_

esta variable siempre tiene el mismo valor para todas las instancias de la clase (objetos\_copia)

esta variable existe incluso sin instanciar la clase

para usarla

class miclase:

varaibleClase = 0 #esta variable es la de clase

#\_\_variableClase = 0

def \_\_init\_\_(self,val=1):

self.\_\_variable = val

miclase.variableClase += 1 #asi se la usa dentro de la clase

print(miclase.\_\_dict\_₎ #aqui veremos que la variable de clase existe sin instanciar la clase

objeto = miclase()

print(objeto.variable\_clase)

#print(objeto.\_miclase\_\_variableClase) #asi se accede a la variable de clase si es privada

DIFERENCIA VARIABLE LOCAL, VARIABLE DE INSTANCIA Y VARIABLE DE CLASE

class miclase:

variableClase = 1 #esta es una variable de clase

def \_\_init\_\_(self, val):

variableLocal = 1 #esta es una variable local de metodo

self.variableInstancia = 1 #esta es una variable de instancia

miclase.variableClase = val #asi se la usa una variable de clase dentro la clase

**DIFERENCIAS ENTRE MODULO Y CLASE**

al usar modulos este es instanciado **una unica vez**

cuando el modulo es importado

entonces tendremos un unico valor para las variables del modulo

al usar clases podemos instanciarlo multiples veces

en cada instancia tendremos una copia de las variable(atributos de la clase

copia1 = mi\_clase()

copia2 = mi\_clase()

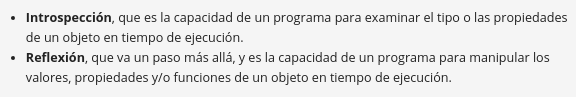
copia3 = mi\_clase()

en cada copia podremos utilizar los atributos de la clase

los atributos de cada copia no interifiere con las otras

igual es necesario importar la clase una vez

**REFLEXION E INTROSPECCION**



**HERENCIA**

isinstance(nombreObjeto , nombreClase)

esta funcion devulve True si el objeto es una isntancia de la clase

devuelve False si no lo es

en caso de preguntar por superclases de superclases

el objeto tambien sera una instancia de ellos

objeto1 is objeto2

#variable1 is variable2

esta funcion devuelve True si el primer argumento **aputna al mismo espacio de memoria**

variable1 = [1]

variable2 = variable1

print(variable1 is variable2) #devolvera True

class Super:

supervar = 1

def \_\_init\_\_(self, nombre):

self.nombre = nombre

def \_\_str\_\_(self):

return "Mi nombre es " + self.nombre + "."

class Sub(Super):

subvar = 2

def \_\_init\_\_(self, nombre):

Super.\_\_init\_\_(self, nombre) #cuando se invoca al contructor de la superclase tambien se

invoca el metodo \_\_str\_\_() automaticamente

obj = Sub("Andy")

print(obj.supervar , obj.subvar)

print(obj)

**las variables de clase tambien son visibles en la herencia**

tambien las **variables de instancia**

nombre\_maestro.\_\_subclasses\_\_()

este metodo **devuelve** una lista con todas las subclases heredadas por nombre\_maestro

ve ejemplo en /programas)2/codigo10.py/

METODO: super()

class Super:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

def \_\_str\_\_(self):

return "Mi nombre es " + self.nombre + "."

class Sub(Super):

def \_\_init\_\_(self, nombre):

super().\_\_init\_\_(nombre) #para invocar a la super clase estamos usando el metodo super()

obj = Sub("Andy")

print(obj)

esta forma de invocacion de superclase nos permite cambiar el nombre de la superclase sin que afecte la funcionabilidad del programa

al invocar al constructor de la superclase no es necesario poner self

**OVERRINDING (**ANULACION)

class Izquierda:

var = "I"

varIzquierda = "II"

def fun(self):

return "Izquierda"

class Derecha:

var = "D"

varDerecha = "DD"

def fun(self):

return "Derecha"

class Sub(Izquierda , Derecha):

pass

obj = Sub()

print(obj.var, obj.varIzquierda, obj.varDerecha, obj.fun())

#salida : I II DD Izquierda

cuando tenemos mas de una superclase y variables o funciones con el mismo nombre, python busca desde la clase mas baja (subclase), despues pasa a las superclases, empezando de izquierda a derecha según haya sido especificado en los argumentos de la sub clase o de manera de herencia jerarquica y devuelve la primera coincidencia

**polimorfismo :**cuando la subclase cambia el comportamiento de la superclase

el metodo de la subclase que cambia el comportamiento de la superclase se llama **virtual**

**ver /programas\_2/codigo8\_1.py**

en el ejemplo vemos que la superclase tiene el metodo girar, pero no sabe como gira cada vehiculo

ahi es donde se instancia las subclases que tiene el feto especifico para girar cada vehiculo en especial, ya que tienen el mismo metodo que la superclase, la cual entrara en accion en lugar del metodo de la super clase

**composicion :** es utilizar otras clases como metodos

**ver /programas\_2/codigo8\_2.py**

la idea de este metodo es **instanciar la superclase dentro la subclase**

asi tenemos un objeto de la superclase dentro la subclase

**no se le pasa las super clases como parametro a la subclase**

en este metodo la superclase es la que sabe los metodos especificos para girar cada tipo de vehiculo

al instanciar la subclase le pasamos **la superclase que invocara como parametro a la subclase**

la clase recivida como parametro en la subclase **se la guarda como variable de instancia,**volviendo se un objeto de instancia

y asi instanciamos la superclase deseada dentro la subclase

**REVISAR** codigo9.py de /programas\_2/

**GENERADORES Y CIERRES**

los generadores son objetos de tipo generador, el resutlado de un generador son solo numeros subsecuentes producidos por el generador,

en realidad los valores resultantes del generador no estan guardados en ningun lado

GENERADORES EN FORMA DE CLASE

class miGenerador:

def \_\_init\_\_(self):

def \_\_iter\_\_(self):

def \_\_next\_\_(self):

for i in miGenerador(10):

print(i)

cuando utilizamos una clase **dentro de un iterador** tenemos que definir las funciones \_\_iter\_\_ y \_\_next\_\_

la funcion \_\_iter\_\_() **define la clase como un generador**

en caso de no deifnirla y usamos la clase dentro un iterador (ciclo for) dara error, ya que no reconocera la clase como un generador

la funcion \_\_next\_\_ **es el generador propiamente dicho**, es la funcion que se repetira las veces sean necesarias, aquí es donde se debe poner lo que queremos que se genere

**importante :** tenemos que poner una condicion que detenga al generador, de lo contrario generara hasta el infinito

**ver ejemplo en /programas\_2/codigo11.py**

GENERADORES EN FORMA DE FUNCION- PALABRA RESERVADA : yield

def fun(n):

for i in range(n):

return i

en el ejmplo naterior no funcionaria como se esperaria

el ciclo nunca se termianria ya que al llegar a la linea return se saldria de la funcion

en caso de querer volver a acceder a la funcion el ciclo se reinicia

def fun(n):

for i in range(n):

yield i

for v in fun(5):

print(v)

lista = [t for t in fun(3)] #se lo puede usar en listas de commprension

lista2 = list(fun(3)) #podemos invocarlo dentro una nueva lista

la palabra yield funciona igual que return pero este almacena el ultimo valor que se especifica para i

yield es un objeto generador, entonces no se debe invocar especificamente

**revisar ejemplo en /programas\_2/codigo11\_2.py**

los generadores en realidad no existen como valores, **son valores subsecuentes producidos po el generador uno por uno**

GENERADORES EN FORMA DE LISTA DE COMPRENSION

genr = (1 if x % 2 == 0 else 0 for x in range(10))

asi de declara un generador en forma de lista de compresion, y lo podemos comprobar preguntando por su tipo

**ver ejemplo en /programas\_2/codigo11\_3.py**

FUNCION: lambda

la funcion lambda nos permite declarar funciones invisibles

**ejemplo 1:**

dos = lambda : 2

cuadrado = lambda x : x \*x

potencia = lambda x , y : x \*\* y

for a in range (-2 , 3):

print(cuadrado(a), end = ‘ ‘)

print(potencia(a , dos()))

las funciones lambda se las puede declarar con su nombre, pero la parte invisible se la aplica en el siguiente ejemplo

def imprimirfuncion(args , funcion):

for x in args:

print(‘f(‘, x , ‘)=’ , funcion(x) , sep = ‘’)

imprimitfuncion( [x for x in range(-2,3)] , lambda x : 2\*x\*\*2 – 4\*x +2)

en este otro ejemplo utilizamos la funcion lambda **para pasar una funcion sin nombre,** la cualno volveremos a usar en otro lugar entonces no es necesario guardar la funcion en una variable

en resumen sirve para declarar funciones sin nombre que la vamos a usar solo en el momento

**ver otro ejemplo en /programas/codigo12.py**

CIERRES

un cierre es cuando **definimos una funcion dentro de otra** y retornamos la funcion interior

al instanciar la funcion exterior el cierre creado es una “copia de la funcion interior”

este cierre de la funcion interior tiene acceso a todas las variables de la funcion exterior a pesar de que la funcion exterior ya no existe