

clase_Python_cont_2021

September 1, 2022

1 Este material está sacado de la cursada 2021

2 Proyecto de Software

3 Nos habíamos quedado en ...

```
[ ]: import string
import random

letras = string.ascii_lowercase
letras += string.ascii_uppercase
letras += string.digits

def generar_clave(largo_clave, todo_minusculas = True):
    clave = ''
    for c in range(largo_clave):
        clave += random.choice(letras)
    if todo_minusculas:
        return clave.lower()
    else:
        return clave

mi_clave = generar_clave(8)
print(mi_clave)
```

4 Expresiones lambda

```
lambda a, b=1: a*b
```

Es equivalente a:

```
def producto(a, b=1):
    return a*b
```

Más info [sobre lambda](#)

```
[ ]: f = lambda a, b=1: a*b
f(2,3)
```

5 Veamos algo más interesante

```
[ ]: lista = [lambda x: x * 2, lambda x: x * 3]
      param = 4
      for accion in lista:
          print(accion(param))
```

6 Usamos lambda: map y filter

```
[ ]: lista = [1, 2, 3, 4, 5, 6, 7]

      secuencia = map(lambda x: x*x, lista)

      #list(secuencia)
      #secuencia = filter(lambda x: x%2 == 0, lista)
      for x in secuencia:
```

7 Manejo de excepciones

- ¿Qué acción se toma después de levantada y manejada una excepción? ¿Se continúa con la ejecución de la unidad que lo provocó o se termina?
- ¿Qué sucede cuando no se encuentra un manejador para una excepción levantada?
- ¿Qué excepciones predefinidas existen?
- ¿Podemos levantar en forma explícita una excepción?
- ¿Podemos crear nuestras propias excepciones?

8 En Python: try - except

```
try:
    sentencias
except excepcion1, excepcion2:
    sentencias
except excepcion3 as variable:
    sentencias
except:
    sentencias
else:
    sentencias
finally:
    sentencias
```

9 Veamos un ejemplo

```
[ ]: dic = {1:"Charly Alberti", 2:"Gustavo Ceratti", 4:"Zeta Bosio"}

for clave in range(1,6):
    try:
        print(f"{clave} - {dic[clave]}")
    except (KeyError):
        print("Clave incorrecta.")
#print (dic)
```

9.1 Python FINALIZA el bloque que levanta la excepción

10 Excepciones: las cláusulas else y finally

```
[ ]: dic = {1:"Charly García", 2:"Pedro Aznar", 4:"David Lebón", 5:"Oscar Moro"}
try:
    for x in range(1,4):
        print (dic[x])
except (KeyError):
    dic[x] = 'NUEVO'
else:
    print ('Se recorrió el diccionario y NO se agregaron valores')
finally:
    print('Ha FINALIZADO la ejecución del modulo')
print ("El diccionario al final del proceso es: ", dic)
```

11 Podemos levantar explícitamente una excepción

```
[ ]: try:
    raise NameError('Hola')
except NameError:
    print("mmm")
    raise
```

Más información en [documentación oficial sobre excepciones](#)

11.1 ¿Cómo es la búsqueda del manejador de una excepción?

```
[ ]: dicci = {0:"Led Zeppelin", 2:"Deep Purple", 3:"Black Sabbath"}
y = 9
try:
    print('Vamos a entrar al bloque TRY interno')
    try:
        for x in range(1,6):
            print (dicci[z])          # OJO que estamos usando la variable z
```

```

except KeyError:
    dicci[x] = 'Agregado'

y = y + 1
print(f"El valor de y es {y}")
except NameError:
    print('OJO! Se está usando una variable que no existe')

print('Se sigue con las siguientes sentencias del programa')

```

- ¿Se ejecutaron las líneas 11 y 12?

12 ¿Y ahora?

```

[ ]: dicci = {0:"Led Zeppelin", 2:"Deep Purple", 3:"Black Sabbath"}
y = 9
try:
    print('Vamos a entrar a otro bloque TRY')
    try:
        for x in range(1,6):
            print (dicci[z])      # OJO que estamos usando la variable z
    except KeyError:
        dicci[x] = 'Agregado'
    finally:
        y = y + 1
        print(f"El valor de y es {y}")
        print("Este texto no se imprime nunca!!!")
except NameError:
    print('OJO! Se está usando una variable que no existe')

print('Se sigue con las siguientes sentencias del programa')

```

12.1 Y.. ¿qué sucede si la excepción se produce en una función?

- Busca dinámicamente.

```

[ ]: def retornar_elemento(x):
    dicci = {0:"Led Zeppelin", 2:"Deep Purple", 3:"Black Sabbath"}
    try:
        return dicci[x]
    except NameError:
        x = 0

print('Este código sirve para mostrar propagación dinámica.')
elem = int(input('Ingresá una clave para acceder al diccionario: (999 para_
↳finalizar) '))
while elem!=999:

```

```

try:
    print (f"El valor del elemento: {elem} es {retornar_elemento(elem)}")
except KeyError:
    print ('OJO! Entraste una clave inexistente. Probá de nuevo!')
elem = int(input('Ingresá clave para acceder al diccionario: (999 para_
↳finalizar) '))

```

13 ¿Qué sucedió?

- La excepción `KeyError` se levantó dentro de la función `retornar_elemento`.
- Al no encontrar un manejador para esa excepción en la función ...
- Busca dinámicamente a quién llamó a la función.
- Si no encuentra un manejador... entonces termina el programa.

14 ¿Es posible acceder a la información de contexto de la excepción?

```

[ ]: dicci = {0:"Led Zeppelin", 2:"Deep Purple", 3:"Black Sabbath"}

try:
    print ('Entramos al bloque try')
    for x in range(0,4):
        print (dicci[x])
    print('Continuamos con el proceso..')
except KeyError as exc:
    dicci[x] = 'NUEVO'
    datos_exc = exc
    import sys
    print(sys.exc_info())
datos_exc

```

15 Objetos en Python

```

[ ]: class Jugador():
    "Define la entidad que representa a un jugador"

    def __init__(self, nic, juego, plataforma="PS4"):
        self._nick = nic
        self._juego = juego
        self._plataforma = plataforma
        self._puntaje = 0

    def get_nick(self):
        return self._nick

```

```

def set_nick(self, nuevo_nick):
    self._nick = nuevo_nick

jugador = Jugador('Nico', 'FIFA')
jugador.set_nick("Lionel")
print(jugador.get_nick())

```

- El método `init()` se invoca automáticamente al crear el objeto.
- Es una buena práctica definir los docstrings
- `self`
- Público y privado.

16 Códigos secretos

```

[ ]: class CodigoSecreto:
    ''' Textos con clave '''

    def __init__(self, texto_plano, clave_secreta):
        self.__texto_plano = texto_plano
        self.__clave_secreta = clave_secreta

    def desencriptar(self, clave_secreta):
        '''Solo se muestra el texto si la clave es correcta'''
        if clave_secreta == self.__clave_secreta:
            return self.__texto_plano
        else:
            return ''

texto_secreto = CodigoSecreto("Lenguaje Python", "Bruce Springsteen")
print(texto_secreto.desencriptar("Bruce Springsteen"))
#texto_secreto.__texto_plano
print(texto_secreto._CodigoSecreto__texto_plano)

```

¿Qué pasa si tenemos quiero imprimir desde fuera de la clase: `**texto_secreto.__texto_plano**?`

Probemos: `print(texto_secreto._CodigoSecreto__texto_plano)`

17 Los símbolos de subrayados

En este artículo se describe el [uso de underscores en Python](#)

17.1 Algunos métodos especiales

- `__str__`
- `__lt__`, `__gt__`, `__le`, `__ge`
- `__eq__`, `__ne`

```
[ ]: class Jugador:
    " Esta clase representa a un jugador"
    def __init__(self, nick, juego):
        self._nick = nick
        self._juego = juego
        self._puntaje = 0

    def get_nick(self):
        return self._nick

    def __str__(self):
        return ("{} juega al {}".format(self._nick, self._juego))

    def __lt__(self, otro):
        return (self._nick < otro.get_nick())

    def __eq__(self, otro):
        return (self._nick == otro.get_nick())

    def __ne__(self, otro):
        return (self._nick != otro.get_nick())

nico = Jugador("Nico", "FIFA2020")
kun = Jugador("Kun", "CSGO")
print(nico)
print(nico < kun)
print(nico if nico == kun else kun)
```

18 Jugadores de LOL y FIFA

- Un jugador de LOL “es un” Jugador.
- La clase que hereda se denomina **clase derivada** y la clase de la cual se deriva se denomina **clase base**.

```
[ ]: class Jugador:
    def __init__(self, nombre, juego="Tetris", tiene_equipo=False, equipo=None):
        self.nombre = nombre
        self.juego = juego
        self.tiene_equipo = tiene_equipo
        self.equipo = equipo
    def jugar(self):
        if self.tiene_equipo:
            print (f"{self.nombre} juega en el equipo {self.equipo} al_
↪{self.juego}")
        else:
            print(f"{self.nombre} juega solo al {self.juego}")
```

```

class JugadorDeFIFA(Jugador):
    def __init__(self, nombre, equipo):
        Jugador.__init__(self, nombre, "PS4", True, equipo)

class JugadorDeLOL(Jugador):
    def __init__(self, nombre, equipo):
        Jugador.__init__(self, nombre, "LOL")
        #super().__init__(nombre, "LOL")

nico = JugadorDeFIFA('Nico Villalba', "Basilea")
nico.jugar()
faker = JugadorDeLOL("Faker", "SK Telecom")
faker.jugar()

```

19 ¿e-sports?

)

- Un jugador de FIFA “es un” Jugador, pero también “es un” Deportista.
- Python tiene **herencia múltiple**

20 e-sports en Python

```

[ ]: class Jugador:
    def __init__(self, nombre, juego="Tetris", tiene_equipo= False,
    equipo=None):
        self.nombre = nombre
        self.juego = juego
        self.tiene_equipo = tiene_equipo
        self.equipo = equipo
    def jugar(self):
        if self.tiene_equipo:
            print (f"{self.nombre} juega en el equipo {self.equipo} al
            {self.juego}")
        else:
            print(f"{self.nombre} juega solo al {self.juego}")

class Deportista:
    def __init__(self, nombre, equipo = "Racing"):
        self.nombre = nombre
        self.equipo = equipo
    def jugar(self):
        print (f"Mi equipo es {self.equipo}")

class JugadorDeFIFA(Jugador, Deportista):

```

```

def __init__(self, nombre, equipo):
    Jugador.__init__(self, nombre, "PS4", True, equipo)
    Deportista.__init__(self, nombre, equipo)

class JugadorDeLOL(Deportista, Jugador):
    def __init__(self, nombre, equipo):
        Jugador.__init__(self, nombre, "LOL")
        Deportista.__init__(self, nombre, equipo)

nico = JugadorDeFIFA('Nico Villalba', "Basilea")
nico.jugar()
faker = JugadorDeLOL("Faker", "SK Telecom")
faker.jugar()

```

21 e-sports en Python

- Ambas clases bases tienen definido un método **jugar**: en este caso, se toma el método de la clase más a la **izquierda** de la lista.
- Por lo tanto, es MUY importante el orden en que se especifican las clases bases.
- Más información en [documentación oficial](#)

21.0.1 MRO: Method Resolution Order

```
[ ]: JugadorDeLOL.__mro__
```

22 ¿Qué diferencia hay entre villanos y __enemigos?

```
[ ]: class SuperHeroe():
    villanos = []

    def __init__(self, nombre, alias):
        self._nombre = nombre
        self._enemigos = []

```

- **villanos** es una **variable de clase** mientras que ****_enemigos**** es una **variable de instancia**.
- ¿Qué significa esto?

23 Veamos el ejemplo completo:

```
[ ]: class SuperHeroe():
    """ Esta clase define a un superheroe
    villanos: representa a los enemigos de todos los superhéroes
    """

```

```

villanos = []

def __init__(self, nombre, alias):
    self._nombre = nombre
    self._enemigos = []

def get_nombre(self):
    return self._nombre

def get_enemigos(self):
    return self._enemigos

def agregar_enemigo(self, otro_enemigo):
    "Agrega un enemigo a los enemigos del superhéroe"

    self._enemigos.append(otro_enemigo)
    SuperHeroe.villanos.append(otro_enemigo)

```

```

[ ]: # OJO que esta funcion está FUERA de la clase
def imprimo_villanos(nombre, lista_de_villanos):
    "imprime la lista de todos los villanos de nombre"
    print("\n"+"*"*40)
    print(f"Los enemigos de {nombre}")
    print("*"*40)
    for malo in lista_de_villanos:
        print(malo)

```

```

batman = SuperHeroe( "Bruce Wayne", "Batman")
ironman = SuperHeroe( "Tony Stark", "ironman")

batman.agregar_enemigo("Joker")
batman.agregar_enemigo("Pinguino")
batman.agregar_enemigo("Gatubela")

ironman.agregar_enemigo("Whiplash")
ironman.agregar_enemigo("Thanos")

```

```

[ ]: #imprimo_villanos(batman.get_nombre(), batman.get_enemigos())
#imprimo_villanos(ironman.get_nombre(), ironman.get_enemigos())

imprimo_villanos("todos los superhéroes", SuperHeroe.villanos)

```

24 Propiedades

```
[ ]: class Demo:
    def __init__(self):
        self._x = 0

    def getx(self):
        print("estoy en get")
        return self._x

    def setx(self, value):
        print("estoy en set")
        self._x = value

    def delx(self):
        print("estoy en del")
        del self._x

    x = property(getx, setx)
```

```
[ ]: obj = Demo()
obj.x = 10
print(obj.x)
```

25 La función property()

`property()` crea una propiedad de la clase.

```
property(fget=None, fset=None, fdel=None, doc=None)
```

```
x = property(getx, setx, delx, "x es una propiedad")
```

- Más info: <https://docs.python.org/3/library/functions.html?highlight=property#property>

26 Decoradores

- Un decorador es una función que recibe una función como argumento y extiende el comportamiento de esta última función sin modificarla explícitamente.

26.0.1 RECORDAMOS: las funciones son objetos de primera clase

- ¿Qué significa esto? Pueden ser asignadas a variables, almacenadas en estructuras de datos, pasadas como argumentos a otras funciones e incluso retornadas como valores de otras funciones.

```
[ ]: def decimos_hola(nombre):
    return f"Hola {nombre}!"
```

```
def decimos_chau(nombre):
    return f"Chau {nombre}!"

def saludo_a_Clau(saludo):
    return saludo("Clau")
```

```
[ ]: saludo_a_Clau(decimos_hola)
saludo_a_Clau(decimos_chau)
```

27 ¿Qué podemos decir de este ejemplo?

- Ejemplo sacado de <https://realpython.com/primer-on-python-decorators/>

```
[ ]: def decorador(funcion):
    def funcion_interna():
        print("Antes de invocar a la función.")
        funcion()
        print("Después de invocar a la función.")

    return funcion_interna

def decimos_hola():
    print("Hola!")
```

```
[ ]: saludo = decorador(decimos_hola)
```

- ¿De qué tipo es saludo?

```
[ ]: saludo()
```

28 Otra forma de escribir esto:

```
[ ]: def decorador(funcion):
    def funcion_interna():
        print("Antes de invocar a la función.")
        funcion()
        print("Después de invocar a la función.")
    return funcion_interna

@decorador
def decimos_hola():
    print("Hola!")
```

```
[ ]: decimos_hola()
```

29 Es equivalente a:

```
decimos_hola = decorador(decimos_hola)
```

- +Info
- +Info en español

30 @property

```
[ ]: class Demo:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x

obj = Demo()
obj.x = 10 # Esto dará error: ¿por qué?
print(obj.x)
```

- @property es un **decorador**: una función que recibe una función como argumento y retorna otra función.
- ¿Cuál es la función que estamos aplicando? ¿Y cuál es la que pasamos como argumento?
- No podemos modificar la propiedad x. ¿Por qué?

31 El ejemplo completo

```
[ ]: class Demo:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

obj = Demo()
obj.x = 10
```

```
print(obj.x)
del obj.x
```

32 Métodos de clase

- Se utiliza el decorador `@classmethod`.
- Se usa `cls` en vez de `self`. ¿A qué hace referencia este argumento?

```
[ ]: class SuperHeroe():
    villanos = []

    @classmethod
    def limpiar_villanos(cls, confirmo=False):
        if confirmo:
            cls.villanos = []
        else:
            return cls.villanos

    def __init__(self, nombre, alias):
        self._nombre = nombre
        self._enemigos = []

SuperHeroe.limpiar_villanos()
```

33 Por último: archivos JSON

```
[ ]: import json

archivo = open("ejemplos/bandas.txt", "w")
datos = [
    {"nombre": "William Campbell", "ciudad": "La Plata", "ref": "www.instagram.
↵com/williamcampbellok"},
    {"nombre": "Buendia", "ciudad": "La Plata", "ref": "https://buendia.bandcamp.
↵com/"},
    {"nombre": "Lúmine", "ciudad": "La Plata", "ref": "https://www.instagram.
↵com/luminelp/"}]
json.dump(datos, archivo)
archivo.close()
```

- ¿De qué tipo es la variable `datos`?

```
[ ]: # Ahora accedemos a los datos guardados
import json

archivo = open("ejemplos/bandas.txt", "r")
datos = json.load(archivo)
```

```

datos_a_mostrar =
print(datos_a_mostrar)json.dumps(datos, indent=4)
#print(type(datos))
archivo.close()

```

- ¿De qué tipo de datos? ¿Y datos_a_mostrar?

34 Otro formato: csv

Dataset obtenido de <https://www.kaggle.com/>

```

[ ]: import csv
# abro el dataset

with open('ejemplos/netflix_titles.csv', encoding='utf-8') as data_set:
    reader = csv.reader(data_set, delimiter=',')
    # creo el archivo .csv de salida
    encabezado = next(reader)
    print(encabezado)

    for fila in reader:
        if "Argentina" in fila[5] and fila[7] == "2020":
            print(fila[2])

```

- Iteradores: `next()`

35 Un último ejemplo

```

[ ]: import csv
# abro el dataset

with open('ejemplos/netflix_titles.csv', encoding='utf-8') as data_set:
    reader = csv.reader(data_set, delimiter=',')
    # creo el archivo .csv de salida
    with open('ejemplos/titulos2020.csv', 'w', encoding='utf-8') as salida:
        writer = csv.writer(salida)

        # pongo el encabezado
        writer.writerow(reader.__next__())

        # escribo sólo los títulos estrenados en 2020
        writer.writerows(filter(lambda titulo: titulo[7] == '2020', reader))

```

- Escribimos csv.
- ¿`writer.writerows(filter(lambda titulo: titulo[7] == '2020', reader))`?

36 Nos vemos en la próxima ...