# clase3_3

September 6, 2021

## 1 Proyecto de Software

### 1.1 Cursada 2021

## 2 Nos habíamos quedado en ...

[1]:
```
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, False)
print(mi_clave)
```

```
sTArone7
```

## 3 Expresiones lambda

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

Es equivalente a:

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

Más info sobre lambda

[2]:
```
lambda a, b=1: a*b
```

[2]:
```
<function __main__.<lambda>(a, b=1)>
```

## 4 Veamos algo más interesante

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

```
8
12
```

## 5 Usamos lambda: map y filter

[4]:
```
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)
#list(secuencia)
```

[4]:
```
[1, 4, 9, 16, 25, 36, 49]
```

## 6 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?

## 7 En Python: try - except

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

```
finally:
    sentencias
```

## 8 Veamos un ejemplo

[5]:
```
dic ={1:"Charly Alberti", 2:"Gustavo Ceratti", 4:"Zeta Bosio"}
try:
    for clave in range(1,6):
        print(f"{clave} - {dic[clave]}")
except (KeyError):
    print("Clave incorrecta.")
#print (dic)
```

```
1 - Charly Alberti
2 - Gustavo Ceratti
Clave incorrecta.
```

### 8.1 Python FINALIZA el bloque que levanta la excepción

## 9 Excepciones: las cláusulas else y finally

[6]:
```
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)
```

```
Charly García
Pedro Aznar
Ha FINALIZADO la ejecución del modulo
El diccionario al final del proceso es:  {1: 'Charly García', 2: 'Pedro Aznar',
4: 'David Lebón', 5: 'Oscar Moro', 3: 'NUEVO'}
```

## 10 Podemos levantar explícitamente una excepción

[7]:
```
try:
    raise NameError('Hola')
except  NameError:
    print("mmmm")
```

```
    #raise
```

mmmm

Más información en documentación oficial sobre excepciones

### 10.1 ¿Cómo es la búsqueda del manejador de una excepción?

[8]:
```
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')
```

```
Vamos a entrar al bloque TRY interno
OJO! Se está usando una variable que no existe
Se sigue con las siguientes sentencias del programa
```

- ¿Se ejecutaron las líneas 11 y 12?

## 11 ¿Y ahora?

[9]:
```
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')
```

```
Vamos a entrar a otro bloque TRY
El valor de y es 10
OJO! Se está usando una variable que no existe
Se sigue con las siguientes sentencias del programa
```

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

- Busca **dinámicamente**.

[10]:
```
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) '))
```

```
Este código sirve para mostrar propagación dinámica.
Ingresá una clave para acceder al diccionario: (999 para finalizar) 0
El valor del elemento: 0 es Led Zeppelin
Ingresá clave para acceder al diccionario: (999 para finalizar) 1
OJO! Entraste una clave inexistente. Probá de nuevo!
Ingresá clave para acceder al diccionario: (999 para finalizar) 999
```

## 12 ¿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.

## 13 ¿Es posible acceder a la información de contexto de la excepción?

[11]:
```
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
```

```
Entramos al bloque try
Led Zeppelin
(<class 'KeyError'>, KeyError(1), <traceback object at 0x7f80ee5b70c0>)
```

## 14 Objetos en Python

[12]:
```
class Jugador():
    "Define la entidad que representa a un jugador"

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

```
Lionel
```

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

- Público y privado.

## 15 Códigos secretos

[13]:
```
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)
```

```
Lenguaje Python
```

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

Probemos: print(texto_secreto._CodigoSecreto___texto_plano)

## 16 Los símbolos de subrayados

En este artículo se describe el uso de underscores en Python

### 16.1 Algunos métodos especiales

- ___str___
- ___lt___, ___gt, ___le, ___ge___
- ___eq___, ___ne___

[15]:
```
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)
```

```
Nico juega al FIFA2020
False
Kun juega al CSGO
```

## 17 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**.

[16]:
```
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()
```

```
Nico Villalba juega en el equipo Basilea al PS4
Faker juega solo al LOL
```

## 18 ¿e-sports?

)

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

## 19 e-sports en Python

[17]:
```
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()
```

```
Nico Villalba juega en el equipo Basilea al PS4
Mi equipo es SK Telecom
```

## 20 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 ¿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?

## 22 Veamos el ejemplo completo:

[18]:
```
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)
```

[19]:
```
# 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")
```

[20]:
```
imprimo_villanos(batman.get_nombre(), batman.get_enemigos())
#imprimo_villanos(ironman.get_nombre(), ironman.get_enemigos())

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

```
****************************************
Los enemigos de Bruce Wayne
****************************************
Joker
Pinguino
Gatubela
```

## 23 Propiedades

[21]:
```
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):
        del self._x

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

[22]:
```
obj = Demo()
obj.x = 10
print(obj.x)
```

```
10
```

## 24 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

## 25 Decoradores

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

### 25.0.1 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.

[23]:
```
def decimos_hola(nombre):
    return f"Hola {nombre}!"
```

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

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

[24]:
```
saludo_a_Clau(decimos_hola)
#saludo_a_Clau(decimos_chau)
```

[24]:
```
'Hola Clau!'
```

## 26 ¿Qué podemos decir de este ejemplo?

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

[25]:
```
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!")
```

[26]:
```
saludo = decorador(decimos_hola)
```

- ¿De qué tipo es saludo?

[27]:
```
saludo()
```

```
Antes de  invocar a la función.
Hola!
Después de invocar a la función.
```

## 27 Otra forma de escribir esto:

[28]:
```
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!")
```

[29]:
```
decimos_hola()
```

```
Antes de  invocar a la función.
Hola!
Después de invocar a la función.
```

## 28 Es equivalente a:

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

- +Ínfo
- +Info en español

## 29 @property

[30]:
```
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)
```

```
0
```

- @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?

## 30 El ejemplo completo

[31]:
```
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
```

10

## 31 Métodos de clase

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

[32]:
```
class SuperHeroe():
    villanos = []

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

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

SuperHeroe.limpio_villanos()
```

[32]: []

## 32 Por último: archivos JSON

[1]:
```
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?

[2]:
```
# Ahora accedemos a los datos guardados
import json

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

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

```
[
    {
        "nombre": "William Campbell",
        "ciudad": "La Plata",
        "ref": "www.instagram.com/williamcampbellok"
    },
    {
        "nombre": "Buendia",
        "ciudad": "La Plata",
        "ref": "https://buendia.bandcamp.com/"
    },
    {
        "nombre": "L\u00famine",
        "ciudad": "La Plata",
        "ref": "https://www.instagram.com/luminelp/"
    }
]
```

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

## 33 Otro formato: csv

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

[3]:
```
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])
```

```
['show_id', 'type', 'title', 'director', 'cast', 'country', 'date_added',
'release_year', 'rating', 'duration', 'listed_in', 'description']
```

- Iteradores: **next()**

## 34 Un último ejemplo

[4]:
```
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 titulos estrenados en 2020
        writer.writerows(filter(lambda titulo: titulo[7] == '2020', reader))
```

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

## 35 Nos vemos en la próxima ...