clase_Intro_Python_2022

September 1, 2022

1 Proyecto de Software

1.1 Cursada 2022

2 ¿Qué abordaremos en esta clase?

- Una introducción a Python.
  - Aspectos básicos.
  - POO
- Para más detalle: clase2_1 y clase2_2 de 2021
- Dejamos dos videos más con temas específicos: classe2_3 y clase2_4 de 2021
- Pueden solicitar matricularse en el curso del Seminario 2022.

3 Programación en el servidor

Roadmap

4 Nosotros usaremos Python

4.1 ¿Por qué?

- Es un lenguaje que en los últimos años ha crecido de manera constante.
  - Stack Overflow Trends
  - `https://githut.info/`

5 Documentación y referencias

- Sitio oficial: http://python.org/
- Documentación en español: https://wiki.python.org/moin/SpanishLanguage
- Python Argentina: http://python.org.ar/
- Otras referencias:
  - https://docs.python-guide.org/
  - https://realpython.com/

IMPORTANTE: en los tutoriales y cursos en línea chequear la **versión** de Python.

6 Características del lenguaje

Es un lenguaje de alto nivel, fácil de aprender. Muy expresivo y legible.

```
numero_aleatorio = random.randrange(5)
gane = False
print("Tenés 5 intentos para adivinar un entre 0 y 99")
intento = 1

while intento < 6 and not gane:
    numero_ingresado = int(input('Ingresa tu número: '))
    if numero_ingresado == numero_aleatorio:
        print('Ganaste! y necesitaste {} intentos!!!'.format(intento))
        gane = True
    else:
        print('Mmmm ... No.. ese número no es... Seguí intentando.')
        intento += 1
if not gane:
    print('Perdiste. El número era: {}'.format(numero_aleatorio))
```

- Sintaxis muy clara. **Indentación obligatoria.**

7 Importante: la legibilidad

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- … The Zen of Python

8 Python Enhancement Proposals (PEP)

- Las PEP son documentos que proporcionan información a la comunidad de Python sobre distintas características del lenguaje, novedades en las distintas versiones, guías de codificación, etc.
- La PEP 0 contiene el indice de todas las PEP.
- La PEP 20: el Zen de Python…

9 PEP 8: guía de estilo de codificación

"El código es leído muchas más veces de lo que es escrito" (Guido Van Rossum)

- PEP 8
- Hay guías sobre la indentación, convenciones sobre los nombres, etc.
- Algunos IDE chequean que se respeten estas guías.
- Su adopción es MUY importante cuando se comparte el código.

10 Características del lenguaje (cont.)

- Es **interpretado**, **multiplataforma** y **multiparadigma**
- Posee tipado dinámico y fuerte.
- Tiene un manejo eficiente de estructuras de datos de alto nivel.

11 Estamos usando Jupyter Notebook

[1]:
```
#### Adivina adivinador....
import random
numero_aleatorio = random.randrange(5)
gane = False

print("Tenés 3 intentos para adivinar un entre 0 y 4")
intento = 1

while intento < 3 and not gane:
    numero_ingresado = int(input('Ingresa tu número: '))
    if numero_ingresado == numero_aleatorio:
        print('Ganaste! y necesitaste {} intentos!!!'.format(intento))
        gane = True
    else:
        print('Mmmm ... No.. ese número no es... Seguí intentando.')
        intento += 1
if not gane:
    print('Perdiste. El número era: {}'.format(numero_aleatorio))
```

```
Tenés 3 intentos para adivinar un entre 0 y 4
Ingresa tu número: 2
Mmmm … No.. ese número no es… Seguí intentando.
Ingresa tu número: 2
Mmmm … No.. ese número no es… Seguí intentando.
Perdiste. El número era: 1
```

12 Empezamos de a poco ...

- Las variables no se declaran.

  - Se crean **dinámicamente** cuando se les asigna un valor.
- Pueden cambiar de tipo a lo largo del programa.
  - Python cuenta con **tipado dinámico**
- Las variables permiten referenciar a los objetos almacenados en la memoria.
- Cada objeto tiene asociado **un tipo, un valor y una identidad**.
  - La identidad actúa como un puntero a la posición de memoria del objeto.
- Una vez que se crea un objeto, su identidad y tipo no se pueden cambiar.
- Podemos obtener la identidad de un objeto con la función **id()**.

[2]:
```
a = "hola"
b = a
c = "hola "
print(a, b, c)

print(id(a), id(b))
```

```
hola hola hola
139982987548848 139982987548848
```

13 Sentencia import

[3]:
```
import string
import random
```

[12]:
```
letras = string.ascii_lowercase
num = random.randrange(4)
num
#print(random.choice(letras))
```

[12]:
```
1
```

[13]:
```
from math import sqrt
raiz = sqrt(16)
print(raiz)
```

```
4.0
```

random.choice() vs. sqrt()

¿Por qué no math.sqrt() o choice()?

14 Un poco más ...

[14]:
```
# Se usan triple comillas para cadenas de más de una línea
print("""
    La computadora ha pensado un número en un rango de 0 a 4.
    Tenés 5 intentos para adivinarlo.
    ¿Lo intentás?
""")
```

```
    La computadora ha pensado un número en un rango de 0 a 4.
    Tenés 5 intentos para adivinarlo.
    ¿Lo intentás?
```

[15]:
```
valor = input('Ingresa algo ')
type(valor)
```

```
Ingresa algo 4
```

[15]:
```
str
```

15 Cadenas con formato

[16]:
```
intento = 3
print('Hola {} !!! Ganaste! y necesitaste {} intentos!!!'.format("claudia",␣
 ↪intento))
```

```
Hola claudia !!! Ganaste! y necesitaste 3 intentos!!!
```

[17]:
```
x = 4
print("{0:2d} {1:3d} {2:4d}".format(x, x*x, x*x*x))
```

```
 4  16   64
```

16 Los f-String

- Fueron introducidos a partir de la versión 3.6.
- Ver la PEP 498
- +Info en la documentación oficial
- Una forma más sencilla de usar el format:

[18]:
```
nombre = "Claudia"
f'Hola {nombre} !!! Ganaste! y necesitaste {intento} intentos!!!'
```

[18]:
```
'Hola Claudia !!! Ganaste! y necesitaste 3 intentos!!!'
```

[19]:
```
x = 4
f"{x:2} - {x*x:} - {x*x*x:4}"
```

[19]:
```
' 4 - 16 -   64'
```

[20]:
```
cad1 = "Cadena alineada a izquierda"
cad2 = "Cadena alineada a derecha"
cad3 = "Cadena centrada"
print(f"\n{cad1:<30}\n{cad2:>30}")
print(f"\n{cad3:^30})")
print(f"\n{cad3:*^30}")
```

```
Cadena alineada a izquierda
     Cadena alineada a derecha

       Cadena centrada        )

*******Cadena centrada********
```

17 Importante: ¡la indentación!

[23]:
```
import string
import random
letras = string.ascii_lowercase
letra = random.choice(letras)
if letra  == "A":
    print("Adivinaste")
else:
    print('Mmmm ... No es una A... Seguí intentando.')
```

```
Mmmm … No es una A… Seguí intentando.
```

17.1 Tipos de datos

- Tipos predefinidos: (Built-In Data Types)
  - Números (enteros, flotantes y complejos)
  - Booleanos
  - Cadenas de texto
  - Listas, tuplas, diccionarios y conjuntos.

```
gane = False
texto_1 = 'Adivinaste!'
intento = 1
temperatura = 17.5
```

- ¿Qué nos indica un tipo de datos?

18 Colecciones básicas

[24]:
```
cadena = "The Beatles"
lista = ["John", "Paul", "Ringo", "George"]
tupla = ("John", "Paul", "Ringo", "George")
diccionario = {cadena: lista, 2: tupla}
```

[25]:
```
print(cadena[0])
print(lista[0])
print(tupla[0])
```

```
T
John
John
```

[27]:
```
print(diccionario[cadena])
```

```
['John', 'Paul', 'Ringo', 'George']
```

19 Listas, tuplas, diccionarios

- Mutables e inmutables

```
cadena = "The Beatles"
lista = ["John", "Paul", "Ringo", "George"]
tupla = ("John", "Paul", "Ringo", "George")
diccionario = {cadena: lista, 2: tupla}
```

- ¿Modificamos estas secuencias?

[29]:
```
tupla[0]= "Bruce"
tupla
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [29], in <cell line: 1>()
----> 1 tupla[0]= "Bruce"
      2 tupla

TypeError: 'tuple' object does not support item assignment
```

20 Son todas referencias...

[30]:
```
rock = ["Riff", "La Renga", "La Torre"]
blues = ["La Mississippi", "Memphis"]
musica = rock
rock.append("Divididos")
```

```
musica
```

[30]:
```
['Riff', 'La Renga', 'La Torre', 'Divididos']
```

[31]:
```
print(id(musica))
print(id(rock))
print(id(blues))
```

```
139982497481728
139982497481728
139982497481856
```

20.0.1 Tarea para el hogar...

¿Cómo hacemos para copiar a otra zona de memoria? Probar:

```
musica1 = musica.copy()

musica1 = musica[:]
```

21 Estructuras de control: sentencias condicionales

- if
- if .. else
- if .. elif.. elif.. else
- A if C else B

[33]:
```
criptos = ["DAI", "USDT"]
cripto = "DAI"
tipo_cripto = "estable" if cripto in criptos  else "cambiante"
print(f"{cripto} es {tipo_cripto}")
```

```
DAI es estable
```

IMPORTANTE: Python utiliza la **evaluación con circuito corto**.

[36]:
```
x = 1
y = 0
if True or x/y:
    print("Mmmm raro...")
else:
    print("nada")
```

```
Mmmm raro...
```

22 match

PEP 636 –> Para Python 3.10

[37]:
```
mes = 2
match mes:
    case 1:
        print("Enero")
    case 2:
         print("Febrero")
    case 3:
        print("Ups... Se acabaron las vacaciones!!! :()")
```

```
Febrero
```

[39]:
```
cadena = "vvv"
match cadena:
    case "uno":
        print("UNO")
    case "dos"  | "tres":
        print("DOS O TRES")
    case _:
        print("Ups.. ninguno de los anteriores")
```

```
Ups.. ninguno de los anteriores
```

23 Estructuras de control: iteraciones

- while
- for .. in

[44]:
```
i = 5
while i >0:
    print(i, end="-")
    i -= 1
```

```
5-4-3-2-1-
```

[48]:
```
for num in range(2, 15, 2):
    print(num, end="-")
```

```
2-4-6-8-10-12-14-
```

[49]:
```
dias = ["domingo", "lunes", "martes", "miércoles", "jueves", "viernes",␣
 ↪"sábado"]
for d in dias:
    print(d, end="-")
```

```
domingo-lunes-martes-miércoles-jueves-viernes-sábado-
```

24 Definición por comprensión

[50]:
```
cuadrados = [x**2 for x in range(10)]
print(cuadrados)

pares = [x for x in cuadrados if x % 2 == 0]
print(pares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 4, 16, 36, 64]
```

[51]:
```
dicci = dict([(x, x**2) for x in (2, 4, 6)])
dicci
```

[51]:
```
{2: 4, 4: 16, 6: 36}
```

25 Funciones

[52]:
```
import string
import random

def generar_clave(largo_clave):
    clave = ''
    for c in range(largo_clave):
        clave  += random.choice(letras)
    return clave

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

mi_clave = generar_clave(10)
print(mi_clave)
```

```
pMqIgknqgg
```

26 Un poco más sobre funciones

[53]:
```
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
```

[55]:
```
mi_clave = generar_clave(8)
mi_clave
```

[55]:
```
'01ZZhmW5'
```

- Las funciones pueden tener valores por defecto.
- Estos parámetros siempre se ubican **al final** de la lista de parámetros.
- Más información en documentación oficial sobre funciones

27 ¿Cuándo se evalúan los valores por defecto en los parámetros?

[56]:
```
i = 4
def funcion(x=i):
    print(x)

i = 10
funcion()
```

```
4
```

28 Las funciones como "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.

[57]:
```
def doble(x):
    return 2*x

f = doble
f(10)
```

[57]:
```
20
```

[58]:
```
secuencia = range(10)
dobles = map(doble, secuencia)
for elem in dobles:
    print(elem, end="-")
```

```
0-2-4-6-8-10-12-14-16-18-
```

29 Expresiones lambda

```
lambda x: 2*x
```

Es equivalente a:

```
    def doble(x):
        return 2*x
```

Más info sobre lambda

[59]:
```
f = lambda a, b=1: a*b
f(2,3)
```

[59]: 6

30 Otro ejemplo

[60]:
```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(2)
g = make_incrementor(6)

print(f(42), g(42))
print(make_incrementor(22)(33))
```

```
44 48
55
```

31 Usos comunes

[62]:
```
lista  = [1, 2, 3, 4, 5, 6, 7]

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

secuencia = filter(lambda x: x%2 == 0, lista)
for elem in secuencia:
    print(elem, end="-")
```

```
2-4-6-
```

32 Decoradores

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

[63]:
```
def decimos_hola(nombre):
    return f"Hola {nombre}!"

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

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

[65]:
```
saludo_a_Clau(decimos_hola)
#saludo_a_Clau(decimos_chau)
```

[65]:
```
'Hola Clau!'
```

33 ¿Qué podemos decir de este ejemplo?

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

[66]:
```
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!")
```

[68]:
```
saludo = decorador(decimos_hola)
```

- ¿De qué tipo es saludo?

[69]:
```
saludo()
```

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

34 Otra forma de escribir esto:

[70]:
```
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!")
```

[71]:
```
decimos_hola()
```

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

35 Es equivalente a:

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

- +Ínfo
- +Info en español

36 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?

37 En Python: try - except

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

38 Veamos un ejemplo

[72]:
```
soda = {1:"Charly Alberti", 2:"Gustavo Ceratti", 4:"Zeta Bosio"}
try:
    for clave in range(1,6):
        print(f"{clave} - {soda[clave]}")
except (KeyError):
    print("Clave incorrecta.")
```

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

Python FINALIZA el bloque que levanta la excepción

[73]:
```python
soda = {1:"Charly Alberti", 2:"Gustavo Ceratti", 4:"Zeta Bosio"}
for clave in range(1,6):
    try:
        print(f"{clave} - {soda[clave]}")
    except (KeyError):
        print("Clave incorrecta.")
```

```
1 - Charly Alberti
2 - Gustavo Ceratti
Clave incorrecta.
4 - Zeta Bosio
Clave incorrecta.
```

39 Sobre los restantes aspectos ...

- ¿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?

40 Objetos en Python

[82]:
```python
class Banda():
    "Define la entidad que representa a una banda"

    def __init__(self, nombre, genero="rock"):
        self._nombre = nombre
        self._genero = genero
        self._integrantes = []

    def get_nombre(self):
        return self._nombre

    def agregar_integrante(self, nuevo_integrante):
        self._integrantes.append(nuevo_integrante)
```

[83]:
```python
soda = Banda("Soda Stereo")
soda.agregar_integrante("Gustavo Ceratti")
print(soda.get_nombre())
```

```
Soda Stereo
```

- Es una buena práctica definir los docstrings
- Público y privado.

41 Los símbolos de subrayados

En este artículo se describe el uso de underscores en Python

41.1 Algunos métodos especiales

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

[88]:
```
class Banda():
    """    Define la entidad que representa a una banda   los m
    """

    def __init__(self, nombre, genero="rock"):
        self._nombre = nombre
        self._genero = genero
        self._integrantes = []

    def get_nombre(self):
        return self._nombre

    def agregar_integrante(self, nuevo_integrante):
        self._integrantes.append(nuevo_integrante)

    def __str__(self):
        return (f"{self._nombre} está integrada por {self._integrantes}")

    def __eq__(self, otro):
        return (self._nombre == otro.get_nombre())
```

[89]:
```
soda = Banda("Soda Stereo")
soda.agregar_integrante("Gustavo Ceratti")
soda.agregar_integrante("Zeta")
print(soda)
```

```
Soda Stereo está integrada por ['Gustavo Ceratti', 'Zeta']
```

[90]:
```
soda = Banda("Soda Stereo")
seru = Banda("Seru Giran")
soda==seru
```

[90]:
```
False
```

42 Herencia en Python

[92]:
```
class Musico:
    def __init__(self, nombre, puesto=None, banda=None):
        self._nombre = nombre
        self._tiene_banda = banda!=None
        self._banda = banda
        self._puesto = puesto

    def info(self):
        if self._tiene_banda:
            print (f"{self._nombre}  canta en la banda {self._banda}")
        else:
            print(f"{self._nombre} es solista ")

class Guitarrista(Musico):
    def __init__(self, nombre, banda=None):
        Musico.__init__(self, nombre, "guitarrista", banda)
        self._instrumento = "guitarra"

    def info(self):
        print (f"{self._nombre}  toca {self._instrumento}")

class Vocalista(Musico):
    def __init__(self, nombre, banda=None):
         Musico.__init__(self, nombre, "vocalista")
```

[93]:
```
bruce = Vocalista('Bruce Springsteen')
bruce.info()
brian = Guitarrista("Brian May", "Queen")
brian.info()
```

```
Bruce Springsteen es solista
Brian May  toca guitarra
```

43 Python tiene herencia múltiple

[94]:
```
class Musico:
    def __init__(self, nombre, puesto=None, banda=None):
        self._nombre = nombre
        self._tiene_banda = banda!=None
        self._banda = banda
        self._puesto = puesto
    def info(self):
        if self._tiene_banda:
```

```
            print (f"{self._nombre}  canta en la banda {self._banda}")
        else:
            print(f"{self._nombre} es solista ")

class Guitarrista(Musico):
    def __init__(self, nombre, banda=None):
        Musico.__init__(self, nombre, "guitarrista", banda)
        self._instrumento = "guitarra"

    def info(self):
        print (f"{self._nombre}  toca {self._instrumento}")

class Vocalista(Musico):
    def __init__(self, nombre, banda=None):
         Musico.__init__(self, nombre, "vocalista")
```

[98]:
```
class VocalistaYGuitarrista(Guitarrista, Vocalista):
    def __init__(self, nombre, banda=None):
        Vocalista.__init__(self, nombre, banda)
        Guitarrista.__init__(self, nombre, banda)
```

[99]:
```
mollo = VocalistaYGuitarrista("Ricardo Mollo", "Divididos")
mollo.info()
```

```
Ricardo Mollo  toca guitarra
```

44 A tener en cuenta ...

- MRO "Method Resolution Order"
- Por lo tanto, es MUY importante el orden en que se especifican las clases bases.
- Más información en documentación oficial

[100]:
```
VocalistaYGuitarrista.__mro__
```

[100]:
```
(__main__.VocalistaYGuitarrista,
 __main__.Guitarrista,
 __main__.Vocalista,
 __main__.Musico,
 object)
```

45 Propiedades

[102]:
```
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


    x = property(getx, setx)
```

[103]:
```
obj = Demo()
obj.x = 10
print(obj.x)
```

```
estoy en set
estoy en get
10
```

46 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

47 @property

[105]:
```
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)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [105], in <cell line: 10>()
      7         return self._x
      9 obj = Demo()
---> 10 obj.x = 10 # Esto dará error: ¿por qué?
     11 print(obj.x)

AttributeError: can't set attribute 'x'
```

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

48 El ejemplo completo

[106]:
```
class Demo:
    def __init__(self):
        self._x = 0

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

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

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

```
10
```

49 Métodos de clase

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

[ ]:
```
class Banda():
    generos = set()

    @classmethod
```

```
    def limpio_generos(cls, confirmo=False):
        if confirmo:
            cls.generos =set()
        else:
            return cls.generos

    def __init__(self, nombre, genero="rock"):
        self._nombre = nombre
        self._genero = genero
        self._integrantes = []
        Banda.generos.add(genero)
```

[ ]:
```
soda = Banda("Soda Stereo")
```

[ ]:
```
Banda.limpio_generos()
```

50 Recursos sobre el Seminario de Python

Sitio público de la materia: https://python-unlp.github.io/2022/

Más específicamente: - Guías de estilo - Buenas prácticas

51 Nos vemos en la próxima ...