Programación para el análisis de datos

Federico Pousa
fpousa@udesa.edu.ar

Hoy

- Clases

- Estructuras de control
  - If
  - While
  - For
  - Funciones
  - Listas por comprensión

- Errors and Exceptions
- Debugging
- Style Guide for Python Code

Clases

Clases

Python viene con diversos tipos de datos:

- Números enteros y decimales
- Caracteres y Strings
- Diccionarios
- Listas
- …

¿Cómo construimos nuestros propios tipos de datos?

clases

¿Cómo son las clases?

- Tienen un nombre
- Tienen atributos
- Tienen métodos

Clases

Por ejemplo, queremos modelar un alumno, con los atributos nombre, apellido y legajo

```
class Alumno():
    def __init__(self):
        # Puedo definir aca los atributos
        self.nombre = '<aun_no_defini_el_nombre>'
        self.apellido = '<aun_no_defini_el_apellido>'
        self.legajo = -1


# Creo un alumno
alumno_1 = Alumno()

# Imprimo el nombre
print(alumno_1.nombre) # imprime por pantalla: <aun_no_defini_el_nombre>

# Cambio el atributo nombre
alumno_1.nombre = 'María'

print(alumno_1.nombre) # imprime por pantalla: María
```

Clases

Podemos definir un método para que imprima por pantalla nombre y apellido

```
class Alumno():
    def __init__(self):
       # Puedo definir aca los atributos
       self.nombre = '<aun_no_defini_el_nombre>'
       self.apellido = '<aun_no_defini_el_apellido>'
       self.legajo = -1

    def imprimi(self):
        print(self.nombre+' '+self.apellido)

# Creo un alumno
alumno_1 = Alumno()
alumno_1.nombre ='Juan'
alumno_1.apellido = 'Perez'

# Uso el método
alumno_1.imprimi()


# imprime por pantalla: Juan Perez
```

Clases

Supongamos que siempre que creamos un alumno queremos inicializar el nombre y apellido.
Para eso modificamos el método especial __init__ y le agregamos dos parametros

```
class Alumno():
    def __init__(self,el_nombre,el_apellido):
        self.nombre = el_nombre
        self.apellido = el_apellido
        self.legajo = -1

    def imprimi(self):
        print(self.nombre+' '+self.apellido)

# Creo un alumno pasándole ahora dos valores
alumno_1 = Alumno('Pedro','Gonzalez')

# Uso el método
alumno_1.imprimi()
```

Clases

Le agregamos un campo para guardar numeros de telefono (0 o más)

```
class AlumnoConTel():
    def __init__(self,el_nombre,el_apellido):
        self.nombre = el_nombre
        self.apellido = el_apellido
        self.legajo = -1
        self.telefonos = list()

    def imprimi(self):
        print(self.nombre+' '+self.apellido)

    def agregar_telefono(self,numero):
        self.telefonos.append(numero)

    def tiene_telefono(self):
        return ( len(self.telefonos)>0 )

    def imprimi_telefonos(self):
        print(self.telefonos)
```

```
# Creo el alumno
alumno_2 = AlumnoConTel('Lucía' ,'Fernandez')

# Si tiene telefono los imprimo
if not alumno_2.tiene_telefono() :
    print('El alumno no tiene teléfonos')

# Agrego un numero de telefono
alumno_2.agregar_telefono('+5491145433232')

alumno_2.imprimi_telefonos() # muestra ['+5491145433232']
```

Clases

Hagamos lo mismo con herencia

```python
class AlumnoConTelConHerencia(Alumno):
    def __init__(self,el_nombre,el_apellido):
        super().__init__(el_nombre, el_apellido)
        self.legajo = -1
        self.telefonos = list()

    def agregar_telefono(self,numero):
        self.telefonos.append(numero)

    def tiene_telefono(self):
        return ( len(self.telefonos)>0 )

    def imprimi_telefonos(self):
        print(self.telefonos)
```

```python
alumno_heredado = AlumnoConTelConHerencia('Pedrito', 'Martinez')
```

```python
alumno_heredado.nombre
```

```
'Pedrito'
```

```python
alumno_heredado.apellido
```

```
'Martinez'
```

```python
alumno_heredado.imprimi()
```

```
Pedrito Martinez
```

```python
alumno_heredado.tiene_telefono()
```

```
False
```

```python
alumno_heredado.agregar_telefono('bla')
```

```python
alumno_heredado.tiene_telefono()
```

```
True
```

Clases

Las clases en Python son todo un mundo.

No hablamos nada de :

- Herencia multiple
- Métodos de clase
- Etc

Si quieren profundizar más, pueden leer la bibliografía oficial de clases

Python Estructuras de control

Python: Estructuras de control

Hasta ahora, ejecución de arriba hacia abajo

Las **estructuras de control** son herramientas para cambiar esto:

- Condicional (if , if else, if elif , ..)
- Ciclos (while)
- Funciones

Python: Estructuras de control / Condicional

Condicional

Sintaxis:

if CONDICION**:**
    CODIGO

if CONDICION**:**
    CODIGO
else:
    CODIGO

Ejemplo:

```
1.  # if simple
2.  a=3
3.  if (a==3):
4.      print('la variable a es igual a 3')
5.
6.  # if con else
7.  a=4
8.  if (a==3):
9.      print('la variable a es igual a 3')
10. else:
11.     print('la variable a no es igual a 3')
12.  print('termine el if')
```

Output:

```
la variable a es igual a 3
la variable a no es igual a 3
termine el if
```

(Nota: los TABs en verde. **Indentar** es la forma que tiene Python para determinar la sintaxis. Los TABs tambien puenden ser reemplazados por espacios (2 o 4), documentacion)

Python: Logica trivaluada

Supongamos el siguiente problema:

Queremos hacer un código que diga si un entero **a** es divisible por otro entero **b**, e imprima por pantalla si es o no divisible. Si el valor de la variable **b** es 0, el programa deberá imprimir que también es divisible.

Solución 1	Solución 2	Solución 3
1. `if (a % b == 0 ):` 2. `    print('Es divisible')` 3. `else:` 4. `    print('No es divisible')`		

Python: Logica trivaluada

Supongamos el siguiente problema:

Queremos hacer un código que diga si un entero **a** es divisible por otro entero **b**, e imprima por pantalla si es o no divisible. Si el valor de la variable **b** es 0, el programa deberá imprimir que también es divisible.

Solución 1	Solución 2	Solución 3
1. `if (a % b == 0 ):` 2. `    print('Es divisible')` 3. `else:` 4. `    print('No es divisible')`	1. `if (b == 0 or a % b == 0) :` 2. `    print('Es divisible')` 3. `else:` 4. `    print('No es divisible')`	

Python: Logica trivaluada

Supongamos el siguiente problema:

Queremos hacer un código que diga si un entero **a** es divisible por otro entero **b**, e imprima por pantalla si es o no divisible. Si el valor de la variable **b** es 0, el programa deberá imprimir que también es divisible.

Solución 1	Solución 2	Solución 3

`if (a % b == 0 ):`
`    print('Es divisible')`
`else:`
`    print('No es divisible')`
	`if (b == 0 or a % b == 0) :`
	`    print('Es divisible')`
	`else:`
	`    print('No es divisible')`
		`if ( a % b == 0 or b == 0) :`
		`    print('Es divisible')`
		`else:`
		`    print('No es divisible')`

Python: Logica trivaluada

A testear ...

	Solución 1	Solución 2	Solución 3

	`if (a % b == 0 ):`
	`  print('Es divisible')`
	`else:`
	`  print('No es divisible')`
		`if (b == 0 or  a % b == 0) :`
		`  print('Es divisible')`
		`else:`
		`  print('No es divisible')`
			`if ( a % b == 0 or b == 0) :`
			`  print('Es divisible')`
			`else:`
			`  print('No es divisible')`

a=4 b=2	Es divisible	Es divisible	Es divisible
a=5 b=2	No es divisible	No es divisible	No es divisible
a=4 b=0	ERROR	Es divisible	ERROR

¿Qué pasó?

Python: Logica trivaluada

No es lo mismo

```
(b == 0 or  a % b == 0)
```

que

```
( a % b == 0 or b == 0)
```

En la lógica trivaluada, el **indefinido** es un valor de verdad especial. Lo vamos a usar para modelar operaciones no totales, para cortar con la ejecución. Formalmente se define así:

$\neg A$	
T	F
F	T
U	U

$A \wedge B$		B: T	B: F	B: U
A	T	T	F	U
	F	F	F	F
	U	U	U	U

$A \vee B$		B: T	B: F	B: U
A	T	T	T	T
	F	T	F	U
	U	U	U	U

El **orden de las fórmulas** importa en ciertos operadores, a diferencia de lo que sucede en la lógica binaria.

Python: Estructuras de control (volviendo...)

Ya vimos Condicional (if , if else, if elif , ..) ...

Nos queda:

- Ciclos
- Funciones
- Returns

Python: Estructuras de control / Ciclos

Los ciclos son estructuras de control que nos permiten ejecutar repetidas veces un código hasta que cambia una condición. Para entrar al ciclo se computa la condición, si es True, se pasa al cuerpo el ciclo, si es False, se sigue con la instrucciones post ciclo. Cuando se termina el cuerpo del ciclo se salta nuevamente a la condición

Sintaxis:

```
while ( CONDICION):
        codigo...
codigo post while....
```

Ejemplo:

```
1.  contador = 0
2.  while (contador < 3):
3.      print('El numero es: ', contador)
4.      contador = contador + 1
```

El numero es: 0
El numero es: 1
El numero es: 2

```
1.  contador = 0
2.  while (contador < 3):
3.      print('El numero es: ', contador)
4.
```

El numero es: 0
El numero es: 0
El numero es: 0
….. (por siempre, infinitas veces)

Python: Estructuras de control / Ciclos

También existen los FOR, los FOR son formas de iterar sobre elementos de algo que los contenga.

Sintaxis:

```
for e in contenedor:
    codigo...
codigo post for....
```

Ejemplo:

```
for i in [1,2,3,4]:
    print(i)
```

```
Imprime:
    1
    2
    3
    4
```

Python: Estructuras de control / Ciclos

Los “for each” sirven para recorrer contenedores más complejos como los diccionarios

Ejemplos:

```
for key in {“fede”:4, “pepe”:2, “maria”:8}:
    print(key)
```

```
Imprime:
      fede
      pepe
      maria
-----------------------------------------
for key, value in {“fede”:4, “pepe”:2, “maria”:8}.items():
    print(key, value)
```

```
Imprime:
      fede 4
      pepe 2
      maria 8
```

Python: Estructuras de control (volviendo...)

Ya vimos Condicional (if , if else, if elif , ..), Ciclos (while, for) ...

Nos queda:

- Funciones

Python: Estructuras de control / Funciones

Las funciones son porciones de código que se ejecutan cuando se llaman. Pueden tomar parámetros y devolver valores.

Sintaxis:

```
def NOMBRE( param1, param2, …):
    <codigo funcion>
    return <valor de retorno> (opcional)
```

Ejemplo:

```
1.  def saludo_generico():
2.      print('Hola a todos')
3.
4.  def saludo_particular( nombre ):
5.      print('Hola ',nombre)
6.
7.  # Llamo a saludo generico
8.  saludo_generico()
9.
10. # Llamo a saludo_particular
11. saludo_particular('Juan')
12. saludo_particular('Maria')
```

```
Hola a todos
Hola  Juan
Hola  Maria
```

Python: Estructuras de control / Funciones

Las funciones son porciones de código que se ejecutan cuando se llaman. Pueden tomar parámetros y devolver valores.

Sintaxis:

```
def NOMBRE( param1, param2, …):
    <codigo funcion>
    return <valor de retorno> (opcional)
```

Ejemplo:

```
1.  def dame_un_3():
2.      return 3
3.
4.
5.  # Llamo a dame_un_3
6.  un_tres = dame_un_3()
7.
8.  print(un_tres)
```

```
3
```

Python: Estructuras de control / Funciones / Parámetros opcionales

Las funciones en python pueden tener parámetros opcionales que, en caso de no ser pasados en el llamado a la función, toman algún valor predefinido.

Ejemplo:

```
1.  def saludo_generico():
2.      print('Hola a todos')
3.
4.  def
    saludo_particular(nombre='Maria'):
5.      print('Hola',nombre)
6.
7.  # Llamo a saludo generico
8.  saludo_generico()
9.
10. # Llamo a saludo_particular
11. saludo_particular('Juan')
12. saludo_particular()
```

```
Hola a todos
Hola  Juan
Hola  Maria
```

Hay muchas otras cuestiones muy importantes como *args, **kwargs, mezcla de parámetros, etc.

Python: Listas por comprensión

Listas por comprensión: herramienta de python **cómodas** para definir listas mediante algún criterio

Sintaxis:

```
[ objeto for objeto in LISTA if CONDICION ]
```

Ejemplo:

```
1.  lista_numeros = [7,9,2,11,14,32,212]
2.  lista_pares = [ num for num in lista_numeros if num % 2 == 0 ]
3.  print(lista_pares)
4.
5.  lista_pares = []
6.  for num in lista_numeros:
7.      if num % 2 == 0:
8.          lista_pares.append(num)
```

```
[2, 14, 32, 212]
```

Python: Listas por comprensión

¿Son lo mismo? No necesariamente. Si la función dentro del loop no es muy compleja, conviene usar listas por comprensión. Podemos medirlo con (%%timeit en **ipython** en vez de python)

```
In [8]: %%timeit
lista_pares = [ num for num in lista_numeros if num % 2 == 0 ]
```

471 µs ± 8.63 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

```
In [10]: %%timeit
lista_pares = []
for num in lista_numeros:
    if num % 2 ==0:
        lista_pares.append(num)
```

668 µs ± 19.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Hay varios post al respecto: https://towardsdatascience.com/speeding-up-python-code-fast-filtering-and-slow-loops-8e11a09a9c2f

Errors and Exceptions

En python hay dos tipos de errores:

- Errores de sintaxis
- Exceptions

Por ejemplo:

```
In [18]: 3 = 3
  File "<ipython-input-18-79bfd1be65e2>", line 1
    3 = 3
      ^
SyntaxError: can't assign to literal
```

Pero además de los **errores** de sintaxis, podemos escribir programas que sean válidos en su sintaxis pero generen un problema en su ejecución.

Errors and Exceptions

Por ejemplo

```
In [19]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-19-9e1622b385b6> in <module>
----> 1 1/0

ZeroDivisionError: division by zero
```

Otro:

```
In [20]: a+3
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-bf58c781999a> in <module>
----> 1 a+3

NameError: name 'a' is not defined
```

Errors and Exceptions

Para “atrapar” estos estas excepciones tenemos **try**

Sintaxis:

```
try:
        code...
except EXCEPTION_NAME:
        code...
```

Por ejemplo:

```
try:
    1/0
except NameError:
    print('Estoy en name error!')
except ZeroDivisionError:
    print('Estoy en división por cero')
print(Terminé!)
```

Errors and Exceptions

Otro ejemplo:

```
try:
    CODE
except:
    print('Estoy en cualquier exception)
```

Excepciones es todo un mundo, tengan **CUIDADO** en cómo las usan.

Pueden leer más en: https://docs.python.org/3/tutorial/errors.html

Debugging

Nuestros códigos ~~(casi)~~ nunca van a estar bien de una.

Necesitamos herramientas para poder encontrar el error:

- Testing: Unit tests, asserts.
- Debugging: PDB (https://docs.python.org/3.8/library/pdb.html), una herramienta que nos ayuda a inspeccionar el código instrucción por instrucción.

PEP8 Style Guide for Python Code

Cómo escribir códigos lindos: PEP 8 -- Style Guide for Python Code (Python Enhancement Proposal)

- Hay 128392387348423748 23 reglas
- Son reglas que facilitan la lectura, no tienen efecto en el código

Herramientas que nos ayudan a cumplir estándares:

- Linters: Pylint, Flake8, mypy, pycodestyle
- Code formatters: Black, isort, autopep8

Recomendación para empezar a programar bien: acostumbrarse a utilizar un buen IDE como VSCode o PyCharm en donde las diferentes cuestiones mencionadas (linting, formatting, debugging) estén bien setteadas.

Python: Resumen hasta aca

Por ahora vimos:

- Tipos y estructuras de datos:
  - Numeros
  - Strings
  - Bools
  - Listas
  - Diccionario
  - Clases
- Estructuras de control:
  - Condicional: if
  - Ciclos: while
  - Funciones
  - Listas por comprensión
  - Excepciones
- Otros:
  - Debugging
  - Code style

Ya están en condiciones de terminar la práctica 1 y 2 !

Link guia de ejercicios