



Introducción a Python

Rodrigo Maranzana

¿Por qué Python?

- Key: legibilidad del Código.
- Es un lenguaje interpretado, sin traducción directa a código de máquina.
 - Existe un intérprete intermedio, que traduce línea por línea y ejecuta.
 - Es decir, no es compilado (como C, C++, Go, etc..)
 - Trade-off: velocidad de desarrollo vs. costo/eficiencia computacional
- Es dinámico, no hay necesidad de declarar el tipo de las variables.

Import this...

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Aunque lo práctico gana a la pureza.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una —y preferiblemente sólo una— manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés (Guido van Rossum).
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los "namespaces" son una gran idea ¡Hagamos más de esas cosas!.

Herramientas para desarrollar

Lenguaje + IDE (Entorno de Desarrollo Integrado)



⚠ Intermedio

Plataformas configuradas:



✓ Fácil



✓ ✓ Muy Fácil

Variables

- En Python podemos asignar a una variable un objeto, definiendolos a partir del signo =

```
texto = "Hola Mundo"

int_ej = 5

float_ej = 1.3

lista = [1,2,3,4]

tupla = (1,2,4)

diccionario = {"key1" : 5, "key2" : 3 , "key3" : 2 }
```

Imprimir el valor de una variable

- Comando: `print(<variable>)`

```
>>> print(texto)
Hola Mundo
>>> print(int_ej)
5
>>> print(float_ej)
1.3
>>> print(lista)
[1, 2, 3, 4]
>>> print(tupla)
(1, 2, 4)
>>> print(diccionario)
{'key1': 5, 'key2': 3, 'key3': 2}
```

Operaciones matemáticas

- Suma (+), resta (-), división (/), multiplicación(*), potencia (**)

```
>>> 1 + 2000
2001
>>> 25 - 3.2
21.8
>>> 1 / 3
0.3333333333333333
>>> 4.3 * 8.5
36.55
>>> 23 ** 5
6436343
```

Listas

- Dada una lista:

```
lista = [25, 23, 3, 44]
```

- Podemos obtener el valor de un elemento con: Lista[<índice>]

```
>>> lista[0]  
25
```

- Podemos usarlo para modificar el valor de una lista:

```
lista[0] = 33.5
```

- Visualizamos toda la lista. ¿Qué más cambió además del valor?:

```
>>> lista  
[33.5, 23, 3, 44]
```

- Podemos agregar un elemento a la lista:

```
lista_2 = lista + [55.8]
```


Listas

- Podemos agregar un elemento a la lista:

```
lista_2 = lista + [55.8]
```

```
lista.append(55.8)
```

```
>>> lista_2  
[33.5, 23, 3, 44, 55.8]
```

```
>>> lista  
[33.5, 23, 3, 44, 55.8]
```

- ¿Cuál es la diferencia entre una lista y una tupla?

```
a = [2, 3, 5]  
b = (2, 3, 5)
```

```
>>> b[2] = 15  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

- ¡Las tuplas son inmutables!

Listas: tamaño

- Podemos usar el comando `len(<lista>)` para evaluar el tamaño de la lista.

```
a = [35, 34, 54, 33, 21, 100]
```

```
len(a)
```

```
>>> len(a)
```

```
6
```

Variables booleanas y operaciones lógicas

- Booleano: Representación lógica binaria: True / False
- Operaciones lógicas:
 - == (igualdad), != (desigualdad), > (mayor que), < (menor que),
 - >= (mayor o igual que), <= (menor o igual que)
- Conectores lógicos: and (y), or (o)

```
a = 3.8  
b = 2.5
```

```
>>> a == b  
False  
>>> a != b  
True  
>>> a > b  
True  
>>> a >= b  
True  
>>> a < b  
False  
>>> a <= b  
False
```

```
>>> (a >= b) and (a != b)  
True
```

```
>>> (a < b) and (a != b)  
False
```

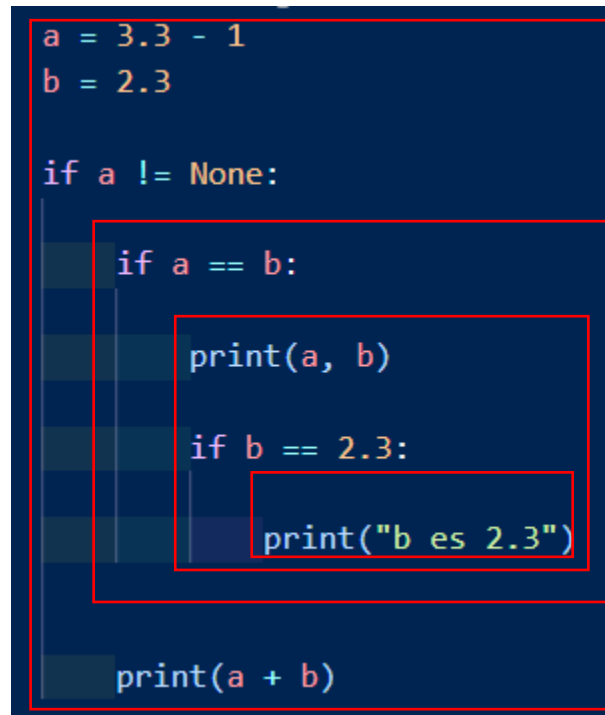
```
>>> (a < b) or (a != b)  
True
```

Indentación

- La jerarquía de ejecución de código en Python se describe con indentaciones:

```
a = 3.3 - 1
b = 2.3

if a != None:
    if a == b:
        print(a, b)
        if b == 2.3:
            print("b es 2.3")
    print(a + b)
```

The diagram illustrates the execution hierarchy of the provided Python code using nested red rectangular boxes. The outermost box encompasses the entire code block. Inside it, a box highlights the first conditional block starting with 'if a != None:'. Within this block, another box highlights the inner conditional 'if a == b:'. Inside the 'if a == b:' block, a box highlights the 'if b == 2.3:' conditional, which in turn contains a box around the 'print("b es 2.3")' statement. Finally, a box at the bottom level highlights the 'print(a + b)' statement, which is not part of any conditional block.

Control de flujo: condicionales if-else

- Usan los resultados de las operaciones lógicas para activar bloques de código:

```
if a == b:  
    print("a igual a b")  
else:  
    print("a distinto de b")
```

`a = 3.3`
`b = 2.3` `a distinto de b`

`a = 3.3 - 1`
`b = 2.3` `a igual a b`

- No es necesario el else:

```
a = 3.3  
b = 2.3  
  
if a == b:  
    print("a igual a b")
```

Control de flujo: condicionales if-else anidados

- Se pueden anidar operaciones de if-else con indentaciones:

```
b = 3.3
a = 2.3

if a == b:
    print("a igual a b")
else:
    if a > b:
        print("a es mayor que b")
    else:
        print("a es menor que b")
        if b <= 4:
            print("b es menor o igual a 4")
```

```
a es menor que b
b es menor o igual a 4
```

Control de flujo: condicionales elif

- Elif es una sentencia que abrevia la ejecución de “else if” anidados:

```
b = 3.3
a = 2.3

if a == b:
    print("a es igual a b")

elif a > b:
    print("a es mayor a b")

elif a < b:
    print("a es menor a b")
```

```
a es menor a b
```

Control de flujo: ciclo “for”

- El ciclo “for” ejecuta una línea de código una cantidad predeterminada de veces.

```
for i in [0, 1, 2, 3, 4]:  
    print("línea número", i)
```

```
línea número 0  
línea número 1  
línea número 2  
línea número 3  
línea número 4
```


Comando “range()”

El comando range crea un rango de valores. Existen variantes:

→ *range(<comienzo>, <fin (sin incluir)>, <paso (opcional)>)*

```
rango = range(0, 5)

list(rango)
```

↓
Convertimos el objeto de
“range” a lista para visualizar.

```
>>> rango = range(0, 5)
>>> list(rango)
[0, 1, 2, 3, 4]
```

Probamos el input “paso”:

```
rango = range(0, 5, 2)

list(rango)
```

```
>>> rango = range(0, 5, 2)
>>> list(rango)
[0, 2, 4]
```

→ *range(<n^a elementos>)* siempre empieza en 0:

```
rango = range(5)

list(rango)
```

```
>>> rango = range(5)
>>> list(rango)
[0, 1, 2, 3, 4]
```

Control de flujo: ciclo “for” con “range”

Podemos hacer más eficiente la implementación:

```
for i in [0, 1, 2, 3, 4]:  
    print("línea número", i)
```

Incorporando la sentencia range():

```
for i in range(5):  
    print("línea número", i)
```

```
línea número 0  
línea número 1  
línea número 2  
línea número 3  
línea número 4
```

Control de flujo: ciclo “while”

Repite la ejecución hasta que se verifica una condición lógica:

```
i = 0
while i != 5:
    print("i distinto de 5. Ejecución paso: ", i)
    i = i + 1
print("i igual a 5")
```

i distinto de 5. Ejecución paso: 0
i distinto de 5. Ejecución paso: 1
i distinto de 5. Ejecución paso: 2
i distinto de 5. Ejecución paso: 3
i distinto de 5. Ejecución paso: 4
i igual a 5

Cuidado con los while infinitos!

Funciones

Bloque que permite encapsular código y reutilizarlo con distintos inputs:

```
def funcion(input1, input2):  
    return input1 + input2  
  
x = funcion(10, 20)  
  
print(x)
```

```
>>> x = funcion(10, 20)  
>>> print(x)  
30
```

Funciones: caso de uso simple, suma ponderada

Tenemos un caso simple de suma ponderada:

```
# ponderadores:  
a = 0.15  
b = 1 - a  
  
# puntajes:  
x = 10  
y = 7  
  
# ratio ponderado:  
rp = a * x + b * y  
  
print(rp)
```

```
>>> rp = a * x + b * y  
>>> print(rp)  
7.45
```

Agregamos una condición sobre los ponderadores.
¿Cómo simplificamos el código?

```
# puntajes:  
x = 10  
y = 7  
  
# Condición:  
if x > 8:  
    a = 0.35  
    b = 1 - a  
    rp = a * x + b * y  
else:  
    a = 0.1  
    b = 1 - a  
    rp = a * x + b * y  
  
print(rp)
```

Generalización de la suma ponderada:

```
def suma_ponderada(a, b, x, y):  
    return a * x + b * y  
  
# puntajes:  
x = 10  
y = 7  
  
# Condición:  
if x > 8:  
    a = 0.35  
    b = 1 - a  
    rp = suma_ponderada(a, b, x, y)  
else:  
    a = 0.1  
    b = 1 - a  
    rp = suma_ponderada(a, b, x, y)  
  
print(rp)
```

```
>>> print(rp)  
8.05
```

Funciones: caso de uso simple, suma ponderada

- Podemos ir más allá y fijar b si consideramos que la suma de los ponderadores tiene que dar siempre 1:

```
def suma_ponderada(a, x, y):  
    b = 1 - a  
    return a * x + b * y  
  
# puntajes:  
x = 10  
y = 7  
  
# Condición:  
if x > 8:  
    a = 0.35  
    rp = suma_ponderada(a, x, y)  
else:  
    a = 0.1  
    rp = suma_ponderada(a, x, y)  
  
print(rp)
```

```
>>> print(rp)  
8.05
```

Funciones: Más de un output

```
def funcion(input1, input2):  
    suma = input1 + input2  
    potencia = input1 ** input2  
  
    return suma, potencia  
  
suma_res, pote_res = funcion(2, 3)  
  
print("suma", suma_res)  
print("potencia", pote_res)
```

```
suma 5  
potencia 8
```

Librerías

- Son colecciones de archivos que contienen módulos de código que se pueden llamar y reutilizar en nuestro proyecto o script.
- Filosofía: “no es necesario inventar nuevamente la rueda”
- Existen personas, grupos de investigación, empresas que crean y mantienen código open-source bajo el esquema de librerías.
- **Siempre considerar el uso de una librería antes de una implementación propia.**
- Ventajas: Consistencia, mantenimiento, fiabilidad, calidad, reproducibilidad...
- Ejemplos:
 - NumPy: <https://numpy.org> - operaciones matemáticas, algebraicas.
 - SciPy: <https://www.scipy.org> - optimización, data science, matemática, ingeniería.
 - Matplotlib: <https://matplotlib.org> - visualización, ploteo.

Librerías

- Para importar una librería:
- Import <librería> as <alias>

```
import numpy as np  
  
import scipy as scp  
  
import matplotlib.pyplot as plt
```

Librerías: Numpy

- Permite hacer tratamiento de matrices, vectores.
- Operaciones más amigables para el usuario.
- Eficientes computacionalmente.
- Convertimos una lista en un vector de Numpy:

```
import numpy as np  
  
vector = np.array([100, 24.3, 25, 40.8])
```

Librerías: Numpy

Ahora tenemos disponibles todos sus métodos, ejemplo suma de vectores:

```
import numpy as np

vector1 = np.array([100, 24.3, 25, 40.8])
vector2 = np.array([300, 1, 35, 38.3])

vector1 + vector2
```

```
>>> vector1 + vector2
array([400. , 25.3, 60. , 79.1])
```

¿Por qué no podemos hacerlo con listas? ¿Por qué

```
lista1 = [100, 24.3, 25, 40.8]
lista2 = [300, 1, 35, 38.3]

lista1 + lista2
```

```
>>> lista1 + lista2
[100, 24.3, 25, 40.8, 300, 1, 35, 38.3]
```

El método "+" concatena listas, no es lo que queremos.

Librerías: Numpy vs. Listas

¿Y cómo sumaríamos con listas? Control de flujo: ciclo for

```
lista1 = [100, 24.3, 25, 40.8]
lista2 = [300, 1, 35, 38.3]

lista3 = [0, 0, 0, 0]

for i in range(0, 4):
    lista3[i] = lista1[i] + lista2[i]

print(lista3)
```

```
>>> print(lista3)
[400, 25.3, 60, 79.1]
>>>
```

Recordar filosofía: “no es necesario inventar nuevamente la rueda”

Comparación de tiempos con vectores de 1000 posiciones:

- Nosotros: 0.0250 segundos
- Numpy: 0.0020 segundos

Librerías: Numpy, trabajo con matrices

```
import numpy as np

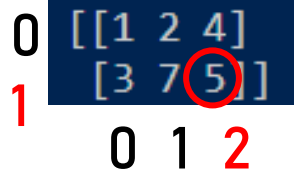
matriz = np.array([[1, 2, 4], [3, 7, 5]])

print(matriz)
```

```
>>> print(matriz)
[[1 2 4]
 [3 7 5]]
```

Cada sublista es una fila de la matriz.

```
matriz[1, 2]
```



A diagram of a 2x3 matrix. The first row is labeled '0' and the second row is labeled '1' in red. The first column is labeled '0', the second '1', and the third '2' in red. The value '5' in the second row, third column is circled in red.

0	[1	2	4]
1	[3	7	5]

Ejemplo: Visualizamos la fila 0 completa:

```
matriz[0, :]
```

```
array([1, 2, 4])
```

Librerías: Matplotlib

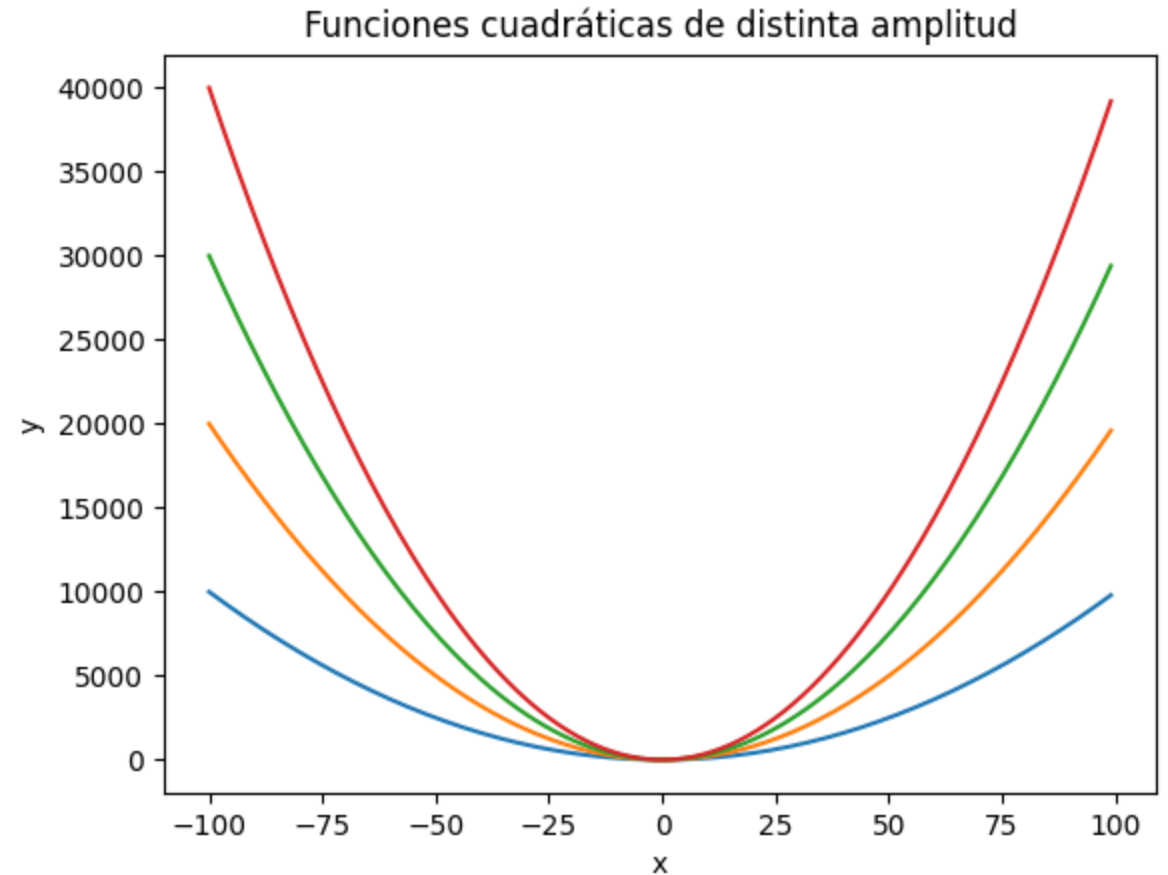
```
import matplotlib.pyplot as plt
import numpy as np

# Operaciones:
a = np.array(range(-100, 100))
b = a**2
c = 2*a**2
d = 3*a**2
e = 4*a**2

# Plots:
plt.plot(a, b)
plt.plot(a, c)
plt.plot(a, d)
plt.plot(a, e)

# Nombre de ejes y título:
plt.xlabel("x")
plt.ylabel("y")
plt.title("Funciones cuadráticas de distinta amplitud")

# Mostrar plot:
plt.show()
```



Consejos para programar en Python

- La **práctica** con un código es la mejor herramienta.
- Un **código propio mal implementado** enseña más que replicar código genérico.
- Python es el **mejor lenguaje para aprender**, similar a pseudo-código.
- Un buen programador no recuerda todo, **sabe qué buscar y dónde**.
 - <https://stackoverflow.com/> es un excelente amigo.