Cómo ser un maestro de Python

Manuel de Castro Caballero
Juan Carlos Gil Díaz

GUI
Grupo Universitario de Informática
Escuela de Ingeniería Informática, Universidad de Valladolid

Hour of Code, 2020
25 de Noviembre de 2020

¿Por qué usar Python?

- ¿Lenguaje de propósito general?
- ¿Multiparadigma?
- ¿Interpretado?
- ¿Débilmente tipado?
- ¿Alta compatibilidad?
- ¿Numerosas herramientas y frameworks?

Por qué uso Python

Programar en Python es *fácil*.

- Puedes hacer **muchas** cosas con relativamente **poco** esfuerzo.
- El intérprete de Python te permite realizar pruebas de partes aisladas de tu código fácilmente.
- El hecho de que sea **fácil** lo hace **rápido** a la hora de programar.

Versiones de Python

- **Python2:** Descontinuadio desde el 01/01/2020. No recomendado, pero algunos sistemas lo siguen utilizando. Última versión: 2.7.18
- **Python3:** Última versión: 3.9.0
  - **Nos centraremos en Python3.**

Instalando Python

- **Python** - Versión por defecto: `https://www.python.org/downloads/`
- **Anaconda** - Versión que incluye múltiples extensiones centradas en la computación científica (pueden instalarse manualmente): `https://www.anaconda.com/products/individual`

Gestor de paquetes

- **pip** - Gestor de paquetes por defecto.
  Ejemplo:

```
\$ pip install jupyterlab
```

- **conda** - Gestor de paquetes de ANACONDA.
  Ejemplo:

```
\$ conda install -c conda-forge jupyterlab
```

- Consola interactiva de PYTHON[1]:

  ```
  \$ python
  ```

  Si hay una versión de PYTHON2 previamente instalada:

  ```
  \$ python3
  ```

  Adecuada para probar fragmentos aislados de código.

[1] Si no funciona, hay que añadir PYTHON a la variable de entorno *path*. (Buscar en Google: "añadir python al path".)

Programando en Python (II)

- Programas completos con extensión `.py`:
  - **PyCharm** (personalmente nunca lo he utilizado).
  - **Cualquier editor de texto**: bloc de notas, Vim, **Sublime Text 3**.

  Para ejecutar:

  ```
  \$ python(3) ./miPrograma.py
  ```

  Python es **interpretado**: ¡no hay que compilarlo!

  Para compensarlo, es muy lento.

Programando en Python (III)

- Utilizando **jupyter lab** (instalación en la diapositiva 7).
  Utilizado en contextos de computación científica: incluir fragmentos de código en documentos explicativos.
  Abrir con:

```
\$ jupyter lab
```

Nuestro primer programa en PYTHON (I)

Vamos a programar "Hola, Mundo!" en PYTHON:

- ¿Sugerencias?

Nuestro primer programa en Python (II)

Vamos a programar "Hola, Mundo!" en Python:

```
print("Hola, Mundo!")
```

Variables (I)

- PYTHON es **debilmente tipado**: "no hay tipos".

```
x = 0
print(x)     # Imprime "0"
x = 0.0
print(x)     # Imprime "0.0"
x = "cero"
print(x)     # Imprime "cero"
x = False
print(x)     # Imrpime "False"
x = [0, 1, 2, 3]
print(x)     # Imrpime "[0, 1, 2, 3]"
```

Variables (II)

- Los tipos no son declarados, pero hay tipos intrínsecos. **Tipado dinámico**.

```
"a" + 0 # Error: "TypeError: can only concatenate
        # str (not "int") to str"
```

- Para averiguar el tipo de una variable: función `type()`

```
type(0)   # "<class 'int'>"
type(0.0) # "<class 'float'>"
type("a") # "<class 'str'>"
```

Variables (III)

- Para *castear* variables a tipos predefinidos:

```
int("0")    # Devuelve 0 de tipo int
int(0.9)    # Devuelve 0 de tipo int
float(0)    # Devuelve 0.0 de tipo float
float("0") # Devuelve 0.0 de tipo float
str(0)      # Devuelve "0" de tipo str
ord("A")    # Devuelve 65 de tipo int
            # (casteo de caracteres a enteros)
chr(65)     # Devuelve "A" de tipo str
            # (casteo de enteros a caracteres)
bool(2)     # Devuelve True de tipo bool
bool(0.0)   # Devuelve False de tipo bool
            # (todo valor distinto de 0 es True)

"a" + str(0) # = "a0"
```

Variables (IV)

- Las variables **no tienen limitaciones de rango**.
- No hay diferencias entre:
  - short/int/long: Solo el tipo entero (int)
  - float/double: Solo el tipo real (float)
  - char/str: Solo el tipo str.

Algunos operadores

- *AND* lógico, *OR* lógico y *NOT* lógico:

```
True and False # False
True or False  # True
not False      # True
```

- Operadores de bit (*AND*, *OR*, *XOR*, *NOT*):

```
1 & 2    # 0
1 | 2    # 3
1 ^ 3    # 2
~1       # -2

# Extensiones por casteos implícitos:
True ^ False    # True
~True           # -2
~False          # -1
```

Algunos operadores (II)

- Exponenciación:

```
2**10    # 1024
```

- Multiplicación de elementos que no son números:

```
"a" * 3          # "aaa"
[0, 1, 2] * 3    # [0, 1, 2, 0, 1, 2, 0, 1, 2]
```

Indentación

IMPORTANTE: indentar el código que va dentro de una estructura de control o función.

- Valen espacios o tabuladores, pero las líneas de código incluidas en el mismo "bloque" deben estar a la misma altura.

Python no utiliza corchetes para crear bloques de código, como sí ocurre ne C o Java.

Estructuras de control de flujo

- **if**, **else** y **elif**

```
if condition:
    # Consecuencia
elif condition2:
    # Alternativa 1
elif condition3:
    # Alternativa 2
else:
    # Alternativa a todas las alternativas
# Código que se ejecuta después del if/elif/else.
```

- **No hay switch**, se debe implementar de forma alternativa (secuencia de elifs).

Estructuras de control de flujo (II)

- **while**:

```
while condition:
    # Bucle
```

- **No hay do-whiles**.
- **for**: Estructura de control un poco particular. No existen de la forma usual, al estilo de JAVA o C. Solo existen del tipo **for-each** (en el taller avanzado se explicará con más detalle). Lo más parecido a un `for` normal es:

```
for i in range(n):
    # Bucle que se repetirá n veces
```

- **Palabras especiales** para bucles:

```
break       # Finaliza prematuramente el bucle
continue    # Pasa a la siguiente iteración
```

Uso de range()

```
range(primero, ultimo + 1, salto_entre_valores)
```

- Por ejemplo, equivalencias con JAVA/C:

```
range(n) = range(0, n) = range(0, n, 1)
(int i = 0; i < n; i++)

range(2, 10, 3)
(int i = 2; i < 10; i += 3)

range(10, 0, -2)
(int i = 10; i > 0; i -= 2)
```

Estructuras de control de errores

- **try-except**:

```
try:
    # Código en el que esperamos un error.
except NameError:
    # Se ejecuta si se detecta una NameError.
    print("Ha ocurrido un NameError!")
except IndexError as e:
    # Se ejecuta si se detecta un IndexError.
    print("Ha ocurrido el siguiente error " + e)
except:
    # Se ejecuta si se detecta cualquier otro error.
else:
    # Se ejecuta si no se detecta ningún error.
    print("No ha habido errores :D")
finally:
    # Se ejecuta al final, en cualquier caso.
    print("Ya no se esperan más errores.")
```

Otras utilidades

- Lanzar errores:

```
raise Exception("Soy un error salvaje.")
```

- Palabra reservada **pass**: No hace nada (literalmente).

```
for i in range(n):
    try:
        # Código en el que puede haber errores.
    except:
        pass # Pasamos de ejecutar si hay un error,
             # continuamos en la siguiente iteración.
```

Funciones

- **Métodos**:

```
def imprimir_hasta(n):
    for i in range(n):
        print(n)
```

- **Funciones**:

```
def factorial(n):
    if (n == 0):
        return 1
    return n * factorial(n - 1)

def varios_valores():
    return 1, 2

x = varios_valores()     # x = (1, 2)
x, y = varios_valores() # x = 1, y = 2
```

Funciones: orden

- **IMPORTANTE**: PYTHON es **interpretado**.
  - El interprete va leyendo el fichero **línea a línea** y ejecutándolas según las lee.
- **Solo se puede llamar a funciones que se han definido anteriormente**, ya que "no existen" en el programa hasta que no se definen.

Entrada/salida

- Lectura de datos

```
entrada = input("Introduzca un número: ") # Devuelve
                                          # una string

n = int(entrada)
```

- Impresión por consola:

```
print("Has introducido " + str(n))
```

Comentarios

- De una linea:

```
# Soy un comentario.
```

- Multilinea:

```
'''Soy un comentario
de varias
líneas.'''
```

~~Arrays~~ Listas

- Python no tiene arrays, tiene **listas**.
- Las listas se diferencian de los arrays:
  - Tienen **tamaño variable**: crecen y decrecen.
  - Pueden almacenar **elementos de distinto tipo**.

```
lista = [] # Lista vacía
lista = [1, 2, 3]
lista = [0, 1.0, True, "3", [4, 5, 6]]
```

Operaciones con listas (I)

```
lista = [0, 1, 2]
```

- **Acceder** a un elemento (por índice):

```
lista[0]  # 0
lista[-1] # 2; los índices negativos se traducen como:
          # "tamaño de la lista + (número indicado)"

# En general, se accede al índice
# ((len(lista) + índice) % len(lista))-ésimo,
# con -len(lisa) <= indice < len(lista)
```

- **Añadir** elemento al final:

```
lista.append(3) # [0, 1, 2, 3]
```

- **Tamaño** de la lista:

```
len(lista) # 3
```

- **Eliminar** un elemento por índice:

```
lista.remove(1) # [0, 2]
```

Operaciones con listas (II)

```
lista = [0, 1, 2, 0, 1, 2]
```

- **Índice** de un elemento:

```
lista.index(0)      # 0
lista.index(0, 1)   # 3
lista.index(3)      # ¡ERROR!
```

- **Comprobar** si existe un elemento:

```
3 in lista          # False
```

- **Contar** elementos:

```
lista.count(0)      # 2
```

- **Añadir** elemento en un índice:

```
lista.insert(3, 3)  # [0, 1, 2, 3, 0, 1, 2]
```

Operaciones con listas (III)

```
lista = [0, 1, 2, 3, 4, 5, 6]
```

- Acceder a una **sublista** (acceso por *slices*):

```
lista[1:3]    # [1, 2]
lista[0:-1:2] # [0, 2, 4]
lista[::-1]   # [6, 5, 4, 3, 2, 1, 0]
# lista[primero:ultimo+1:salto_entre_valores]
# lista[::] = lista[0:len(lista):1]
```

- **Copiar** lista:

```
lista2 = lista[:]
```

- **Extender** con otra lista:

```
lista.extend([7, 8, 9]) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Eliminar y devolver** elemento por índice:

```
lista.pop(3) # [0, 1, 2, 4, 5, 6]; devuelve 3
# lista.pop() = lista.pop(-1)
```

Tuplas

Como las listas, pero **inmutables**: de solo lectura.

- Una vez construidas, no se puden editar.

```
tupla = (1, 2, 3)
```

- Operaciones: `.count()` y `.index()` solamente.

Diccionarios

- Los **diccionarios** son parecidas a las listas, pero en vez de almacenar elementos, se almacenan **pares clave-valor**:

```
diccionario = {}     # Diccionario vacío
```

- Las claves y los valores pueden ser de cualquier tipo:

```
diccionario= {"clave": "valor", 0: False,
              2.0: [0, 1]}
```

- Los valores se **acceden por su clave**:

```
diccionario["clave"]   # Devuelve "valor"
diccionario[2.0] = 5
# diccionario ahora es:
# {"clave": "valor", 0: False, 2.0: 5}
```

Diccionarios II

```
diccionario = { 0: "lunes", 1: "martes", 2: "miercoles" }
```

- Obtener **todas las claves**:

```
diccionario.keys()  # [0, 1, 2]
```

- Obtener **todos los valores**:

```
diccionario.values()  # ['lunes', 'martes', 'miercoles']
```

- Obtener **todos los pares clave-valor** (como tuplas):

```
diccionario.items() # [(0, 'lunes'), (1, 'martes'),
                    # (2, 'miercoles')]
```

- **Eliminar** un par clave-valor:

```
del diccionario[0]  # {1: 'martes', 2: 'miercoles'}
```

Lectura y escritura de ficheros

```
in_ = open("entrada.txt", "r")   # Apertura para lectura
out = open("salida.txt", "w")    # Apertura para escritura

ap = open("añadir.txt", "a")     # Apertura para escritura,
                                 # escribiendo debajo de lo
                                 # que ya haya

io = open("lec_y_esc.txt", "r+")# Apertura para lectura y
                                 # escritura

fichero.close() # Cierre del fichero (tras usarlo)
```

Los modos `"rb"`, `"wb"`, `"ab"`, `"rb+"` se utilizan para trabajar con ficheros **binarios** en vez de textuales.

Funciones de lectura y escritura

- **Lectura** de **todo** un fichero en una string:

```
texto = fichero.read()
```

- **Lectura** de **las líneas** de un fichero en una lista de strings:

```
lista_lineas = fichero.readlines()
```

- **Escritura** de una string a un fichero:

```
fichero.write(string)
```

Importar módulos

Los módulos añaden funcionalidades a nuestros programas. Python cuenta con una gran cantidad de módulos de mucha utilidad, ayudándonos a programar de forma más concisa.

```
import modulo
modulo.funcion()

import modulo as m
m.funcion()

from modulo import funcion
funcion()

from modulo import funcion as f
f()
```

Ejercicios

1. Construir una función que, dado un entero `n`, devuelva la **matriz identidad** de `nxn` dimensiones.
2. Construir una función a la que se le pase un tipo de datos y una lista y devuelva **otra** lista con los **elementos del tipo especificado** de la primera lista.
   - `funcion(int, [0, 1.5, True, "string"])` debe devolver `[0]`

Ejercicios (II)

3. Construir un programa que pida en bucle números al usuario, los almacene en una lista, e imprima por pantalla la **media**, la **moda** y la **desviación típica** del conjunto de números introducido.
   - Media: suma de todos los números entre el número total de números.
     Moda: el número que más se haya repetido (en caso de empate, devolver cualquiera)
     Desviación típica: la raíz cuadrada de la suma de `(numero_i - media)**2 / total_numeros`
   - El bucle puede terminar cuando se introduzca cierto valor especial (0, o algún número negativo), o cuando no se introduzca ningún número, con un `try-except`.

Ejercicios (III)

4. Realizar un programa que modele el login de una página web utilizando un **diccionario**:
   - El programa pedirá en bucle nombres al usuario.
   - Si el nombre introducido no está registrado, pedirá una contraseña y lo registrará en el sistema (diccionario).
   - Si el nombre introducido está registrado, mostrará su contraseña.
     - Otra opción es pedir la contraseña y comprobar que es correcta.

Trucos breves

- Strings con **formato**:

```
x = 5
print(f"Valor de x: {x}")    # "Valor de x: 5"
```

- Funciones de **tratamiento de strings**

```
"a b c d".split(" ")          # ["a", "b", "c", "d"]
"a,b,c,d".replace(",", " ") # "a b c d"
```

- Operador **ternario**:

```
max = a if a > b else b
```

- PYTHON tiene un tipo nativo para los **números complejos**:

```
type(1j)          # "<class 'complex'>"
print(1j**2)      # (-1+0j)
```

Trucos breves (II)

- Un diccionario sin valores es un **conjunto** (`set`):
  - Cada elemento solo se incluye una vez.

    ```
    type({1, 2, 3}) # <class 'set'>
    ```

  - Los conjuntos tienen definidas ciertas operaciones propias, como la unión, la intersección, la diferencia, la comprobación de subconjunto/superconjunto...
- Para acceder a los **argumentos** que se le pasen al programa:

  ```
  from sys import argv

  argv[0]      # Nombre del programa en ejecución
  len(argv)    # Número de argumentos pasados al programa
               # (contando el nombre)
  ```

- Para ejecutar **comandos** arbitrarios en terminal:

  ```
  from os import system

  system("clear")
  ```

For-each

- Los **for** en PYTHON recorren **iterables**.
  - Un iterable es una estructura formada por varios elementos, sobre los que se puede iterar: listas, tuplas, diccionarios...
- Cada iteración, se toma el valor de uno de los elementos del iterable.

```
for e in [0, 1, 2, 3]:
    print(e)
# Imprime en distintas líneas: 0, 1, 2, 3
```

For-each (II)

```
for e in {0: "a", 1: "b", 2: "c"}:
    print(e)
# Imprime en distintas líneas: 0, 1, 2
# (¡Se itera sobre las claves!)

for e in "Bienvenidos a la Hora del Codigo :D".split(" "):
    print(e)
# Imprime:
#  Bienvenidos
#  a
#  la
#  Hora
#  del
#  Codigo
#  :D
```

Comprensiones de listas

Las compresiones de listas presentan una forma concisa de crear listas:

```
l = []
for i in range(100):
    if (i * i) % 2 == 0:
        l.append(i * i)

# Equivale a:
l = [i * i for i in range(100) if (i * i) % 2 == 0]
```

¡Puedes crear listas en una sola expresión (y por tanto en una sola línea)!

Ejemplos de comprensiones de listas

```python
[7 * i for i in range(10)]  # La tabla del 7

[e for e in l1 if e in l2]  # Intersección de dos listas
                            # (l1 y l2)

[print(e) for e in lista]   # Imprimir todos los
                            # elementos de una lista

M = [[0] * 10 for i in range(10)]   # Creación de una
                                    # matriz 10x10

[e for fila in [[0, 1], [2, 3]] for e in fila]
# Comprensión de lista de doble bucle:
# ¡"Aplana" la matriz!: [0, 1, 2, 3]
```

Ejemplo grande de comprensiones de listas

```
[k for i in [[range(3*x + y, 3*(x + 1) + y) for x in range(3)]
    for y in range(0, 27, 9)] for j in i for k in j]
# 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., 26
```

Comprensiones de diccionarios

Las comprensiones también tienen su versión para diccionarios:

```
{chr(e + ord('A')): e for e in range(ord('Z') - ord('A') + 1)}
#  {'A': 0, 'B': 1, 'C': 2, ..., 'Z': 25}
```

Generadores

Podemos entender los generadores como comprensiones de listas "de un solo uso": Solo se puede iterar una vez sobre ellos.

```
gen = (x**2 for x in range(10))
print(generador) # <generator object gen at 0x...>
[print(e) for e in gen] # 0, 1, 4, 9, ..., 81
```

yield

`yield` es una palabra reservada utilizada como `return`, para crear generadores:

```
def mi_generador():
    yield 1
    yield 2
    yield 3

[print(i) for i in mi_generador()]    # 1, 2, 3
```

Cada vez que se llame a la función, se ejecutará hasta el siguiente `yield` que se encuentre. Esto ocurrirá hasta que el generador se considere vacío: cuando la función termina su ejecución sin llegar a ningún `yield`.

Ejemplo de yield

```
def celdas_contiguas(x, y):
    for i in range(-1, 2):
        for j in range(-1, 2):
            if not (i == 0 and j == 0):
                yield (x + i, y + j)

[print(e) for e in celdas_contiguas(1, 1)]
# (0, 0)
# (0, 1)
# (0, 2)
# (1, 0)
# (1, 2)
# (2, 0)
# (2, 1)
# (2, 2)
```

Expansión de argumentos

En una llamada a una función, se puede **expandir** un iterable a sus elementos, para pasarlos como argumentos, utilizando el caracter `*`:

```
def imprimirPunto(x, y, z):
    print(f"Las coordenadas del punto son: {x}, {y}, {z}")

punto = (1, 2, 3)
imprimirPunto(*punto)
# "Las coordenadas del punto son: 1, 2, 3"
```

Argumentos con valores por defecto

Puedes asignar un **valor por defecto a cada argumentos** de una función en PYTHON, siguiendo su declaración de un = y el valor por defecto:

```
from math import sqrt

def ecuacionSegundoGrado(a=1, b=0, c=0):
    return (-b + sqrt(b**2 - 4 * a * c)) / (2 * a)

print(ecuacionSegundoGrado())           # 0.0
print(ecuacionSegundoGrado(2, 2))       # 0.0
print(ecuacionSegundoGrado(1, 2, 1))    # -1.0
```

Se puede cambiar el valor de solo algunos de los argumentos por defecto, indicando cuáles en la llamada a la función:

```
print(ecuacionSegundoGrado(c=-1))       # 1.0
```

Otro ejemplo de argumentos por defecto

Los argumentos por defecto tienen que declararse **después** de los argumentos sin valores por defecto.

```
def ecuacionSegundoGrado(c, b=0, a=1, dos_resultados=False):
    if dos_resultados:
        return (-b + sqrt(b**2 - 4 * a * c)) / (2 * a),
            (-b - sqrt(b**2 - 4 * a * c)) / (2 * a)
    else:
        return (-b + sqrt(b**2 - 4 * a * c)) / (2 * a)

ecuacionSegundoGrado(-1, dos_resultados=True, a=2)
# (0.7071067811865476, -0.7071067811865476)
```

Argumentos variables

Puedes crear funciones que tomen un **número de argumentos variable**.

- No confundir con argumentos por defecto: los argumentos variables **no toman un valor por defecto**.

Los argumentos por defecto se guardan en una lista (parámetro) declarada con un * delante (*args por convención):

```
def multiplicar(a, b, *args):
    f = a * b
    for e in args:
        f *= e
    return f

print(multiplicar(2, 3, 4, 5, 6))    # 720
```

Argumentos variables (II)

Existe una forma de capturar argumentos variables que tengan una **clave**: utilizando `**`. El nombre del parámetro suele ser `**kwargs`, un diccionario:

```
def ejemplo_kwargs(**kwargs):
    [print(k, v) for k, v in kwargs.items()]

ejemplo_kwargs(clave="valor", a=True, b=1)
# clave valor
# a True
# b 1
```

Funciones lambda

- Las **funciones lambda** son funciones anónimas cuya definición es una única expresión. Devuelven el valor de dicha expresión (evaluando sus parámetros).
- Se pueden usar en cualquier lugar en que se puede usar una expresión (asignación a una variable, argumento a otra función...).
- Su forma es:

```
lambda argumentos: expresion
```

- Ejemplo:

```
multiplicar = lambda x, y: x*y

print(multiplicar(2, 5))    # 10
```

map()

`map()` es una función que toma como argumentos una función de un argumento y un iterable y devuelve un objeto map (un iterador), cuyos elementos son el resultado de **aplicar la función a todos los elementos** del iterable:

- Lista de los 10 primeros cuadrados:

```
list(map(lambda x: x**2, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Transformar un csv con números enteros a una matriz:

```
csv = [list(map(int, line.split(",")))
    for line in open("datos.csv").readlines()]
```

Más ejemplos de map()

- Encadenar `maps`:

```
from math import sqrt

map(sqrt, map(abs, listaNumeros))
```

- Mapa sobre una **matriz**:

```
map(lambda l: map(sqrt, l), listaNumeros)
```

zip()

`zip()` es una función que toma 0 o más iterables por argumentos y devuelve un objeto zip (un iterador) cuyos elementos son tuplas conteniendo los elementos de sus argumentos.

```
list(zip(range(3), range(0, 6, 2), range(100)))
# [(0, 0, 0), (1, 2, 1), (2, 4, 2)]
```

reduce()

`reduce()` es una función del módulo `functools`, que toma como argumentos una función de dos argumentos y un iterable, y devuelve el **valor** de aplicar dicha función a todos los elementos del iterable, en secuencia.

- Suma de los 100 primeros números:

```
from functools import reduce

suma100 = reduce(lambda a, b: a + b, range(100))
# 4950
# Equivalente a sum(range(100))
```

- Máximo de una lista:

```
maximo = reduce(lambda a, b: a if a > b else b, lista)
```

Objetos

La forma general de un objeto en Python es:

```python
class Persona:
    nombre = ""

    # Constructor
    def __init__(self, nombre, dni):
        self.nombre = nombre
        self.dni = dni

    def saludo(self):
        return "Buenos dias"

    @staticmethod
    def sumar2y2():
        return 4
```

```python
# Herencia:
class Informatico(Persona):
    # Constructor
    def __init__(self, nombre, username, dni):
        Persona.__init__(self, nombre, dni)
        #super().__init__(self, nombre, dni)
        self.usr = username

    def saludo(self):
        return "Hello, world!"
```

```python
alice = Persona("Alice", "00000001R")

print(alice.saludo())       # Buenos días
print(alice.sumar2y2())     # 4
print(Persona.sumar2y2())   # 4
```

```python
bob = Informatico("Bob", "xX_BobG4m3r_Xx",
    "00000007F")
print(bob.saludo())         # Hello, world!
print(bob.sumar2y2())       # 4
```

Conceptos clave de los objetos

- `self` se usa (por convención) para referenciar a la **instancia concreta** sobre la que se ejecutan funciones o accede a atributos. Es el **primer parámetro** de todas las funciones.
  - Salvo en casos especiales, **el valor de `self` se omite de las llamadas a funciones**.
- `__init__()` es el constructor de objetos.
- Los atributos se pueden definir dentro de la clase, pero no hace falta. Se pueden añadir "sobre la marcha" dentro de las funciones de clase (incluyendo el constructor).
- Los métodos estáticos se **decoran** con `@staticmethod`.
- Puedes **privatizar** miembros haciendo que empiecen por `__`.
  - No es una privatización real. Se sigue pudiendo acceder a cualquier miembro privado: `obj._Clase__miembro` (a esto se le llama *name mangling*).
  - Por ejemplo, para acceder al metodo privado `__privado()` definido en `Persona` de la instancia `bob`: `bob._Persona__privado()`
  - Los miembros privados **no se heredan**.

Definiendo operaciones sobre objetos

- Los operadores de PYTHON, al igual que ciertas funciones estándar, llaman a ciertas **funciones específicas de los objetos** para realizar sus operaciones.
- Al igual que en JAVA puedes definir `.equals()` o `.compareTo()` para definir ciertas operaciones básicas sobre los objetos, en PYTHON, se puede hacer lo mismo.

Mapa básico de operadores

Referencia: https://docs.python.org/3/reference/datamodel.html

El operador:	Equivale a:
`obj = Clase()`	`obj.__new__(Clase)`
`obj1 < obj2`	`obj1.__lt__(obj2)`
`obj1 <= obj2`	`obj1.__le__(obj2)`
`obj1 == obj2`	`obj1.__eq__(obj2)`
`obj1 != obj2`	`obj1.__ne__(obj2)`
`obj1 > obj2`	`obj1.__gt__(obj2)`
`obj1 >= obj2`	`obj1.__ge__(obj2)`
`hash(obj)`	`obj.__hash__()`
`str(obj)`	`obj.__str__()`
`bool(obj)`	`obj.__bool__()`
`... # Más casteos`	`... # obj.__tipo__()`
`obj.attr`	`obj.__getattribute__("attr")`
`obj.attr = x`	`obj.__setattr__("attr", x)`
`obj[x]`	`obj.__getitem__(x)`
`obj[x] = y`	`obj.__setitem__(x, y)`
`obj(x)`	`obj.__call__(x)`
`len(obj)`	`obj.__len__(x)`
`x in obj`	`obj.__contains__(x)`

Mapa básico de operadores (II)

El operador:

```
obj1 + obj2
obj1 - obj2
obj1 * obj2
obj1 @ obj2
obj1 / obj2
obj1 // obj2
obj1 % obj2
obj1**obj2 #pow(obj1, obj2)
obj1 << obj2
obj1 >> obj2
obj1 & obj2
obj1 ^ obj2
obj1 | obj2
obj1 += obj2
... # op=
-obj
+obj
abs(obj)
~obj
```

Equivale a:

```
obj1.__add__(obj2)
obj1.__sub__(obj2)
obj1.__mul__(obj2)
obj1.__matmul__(obj2)
obj1.__truediv__(obj2)
obj1.__floordiv__(obj2)
obj1.__mod__(obj2)
obj1.__pow__(obj2)
obj1.__lshift__(obj2)
obj1.__rshift__(obj2)
obj1.__and__(obj2)
obj1.__xor__(obj2)
obj1.__or__(obj2)
obj1.__iadd__(obj2)
... # obj.__iop__
obj.__neg__()
obj.__pos__()
obj.__abs__()
obj.__invert__()
```

Aplicando lo aprendido

- Se enlaza un fichero del *Advent of Code* de 2019:
  Ejemplo de input del día 14.
- El fichero representa reacciones entre elementos para producir otro elemento:
  - "Por ejemplo, la reacción `1 A, 2 B, 3 C => 2 D` significa que exactamente 2 unidades del elemento `D` pueden producirse consumiendo exactamente `1 A`, `2 B` y `3 C`."
    - Cada reacción **produce un único tipo de elemento**.
- **Construir un diccionario con el que poder consultar las reacciones indicadas por el fichero.**
  - Los elementos producidos podrían ser las claves, y los reactivos los valores.
  - Por ejemplo:

```
{(2, "D"): [(1, "A"), (2, "B"), (3, "C")]}
```

Mi solución

```
reactions = {(int(e.split(" ")[0]), e.split(" ")[1]):
    list(zip((int(n) for n in s.split(" ")[0::2]), s.split(" ")[1::2]))
    for s, e in map(lambda x: tuple(x[:-1].replace(", ", " ").split(" => ")),
    open("transformations.txt").readlines())}
```

Todo esto es **una única línea** (237 caracteres).

NumPy

- NumPy es un módulo de Python de código libre, cuyo propósito es facilitar la **computación numérica** en Python.
- Incluye **funciones** de gran utilidad para ámbitos relacionados con el **álgebra lineal** (vectores, matrices, etc.).
- Usada en múltiples campos de la **computación científica**:
  - Machine learning.
  - Computación cuántica.
  - Computación estadística.
  - Procesamiento de señales.
- En su **página web** ofrecen documentación y tutoriales para aprender a utilizar el módulo de forma fácil.

```
import numpy as np
```

Arrays de NumPy

La **estructura principal** de NumPy es el `ndarray` (*n-dimensional array*), normalmente referido como array.

- Al contrario que las listas, **solo puede almacenar un único tipo de elementos** (normalmente números).
- Los arrays de NumPy son más rápidos y utilizan menos memoria que las listas de Python. La **optimización** es una prioridad principal.

Inicialización de arrays de NumPy

```python
# Casteando iterables:
a = np.array([1, 2, 3], dtype=float)
print(a)     # [1. 2. 3.]

# Otros inicializadores:
np.zeros(3)          # [0. 0. 0.]
np.ones(3)           # [1. 1. 1.]
np.empty(3)          # Valores aleatorios. ¡Rellenar!
np.arange(4)         # [0 1 2 3]
np.arange(2, 10, 2) # [2 4 6 8]

np.eye(2)    # "eye" representa "I": la matriz identidad
             # [[1 0]
             #  [0 1]]
```

Atributos de los arrays

- Número de **ejes** (dimensiones).

  `ndarray.ndim`

- **Dimensiones** del array, como tupla (tamaño del array en cada dimensión).

  `ndarray.shape`

- Número total de **elementos**

  `ndarray.size`

- **Tipo** de los elementos del array (encapsulado en un objeto).

  `ndarray.dtype`

Operaciones con arrays

- **Redimensionar** un array:

```
np.arange(6).reshape(2, 3)
#np.arange(6).reshape((2, 3)) # ¡Funciona igual!
# [[0 1 2]
#  [3 4 5]]
```

- **Acceder** a elementos de un array (por índices o slices):

```
a = np.arange(16)

a[3]     # 3
a[(a > 10)]  # [11 12 13 14 15]
a.reshape((4, 4))[2, 2] # 10
a.reshape((4, 4))[1::2, :]  # [[ 4  5  6  7]
                            #  [12 13 14 15]]
```

Operaciones matemáticas con arrays

- Las operaciones básicas realizadas sobre arrays, se realizan sobre **todos los elementos** del array.

```
a = np.range(5) # [0 1 2 3 4]

a + 5    # [5 6 7 8 9]
a // 2   # [0 0 1 1 2]
a**2     # [0 1 4 9 16]
```

- Si **los dos operandos son arrays**, se suman los elementos correspondientemente (deben tener la misma forma):

```
a - np.arange(5, 0, -1) # [-5 -3 -1 1 3]
```

- También, se define la multiplicación de matrices entre arrays:

```
a @ np.arange(5, 10)     # 80
```

Otras funciones de utilidad

```
a = np.arange(16).reshape(4, 4)

a.min()      # 0
a.max()      # 15
a.sum()      # 120
a.cumsum()   # Suma acumulada para cada elemento:
             # [0 1 3 6 10 15 21 ...]

a.min(axis=0)   # [0 1 2 3]
a.max(axis=1)   # [3 7 11 15]
```

Matrices de NumPy

Las `matrix` de NumPy heredan de `ndarray`, por lo que tienen los mismos atributos y métodos. Sin embargo, existen 6 diferencias fundamentales con los arrays:

- Pueden **crearse a partir de strings**, siguiendo una sintaxis similar a la de Matlab.
- Siempre son **dos-dimensionales**.
- La multiplicación por defecto (*) es **multiplicación matricial**.
- La potenciación de matrices equivale a **elevar la matriz a la potencia** especificada (en vez de cada uno de sus elementos).
- La **"prioridad" de las matrices es más alta** que la de los arrays, por lo que las operaciones entre arrays y matrices devuelven matrices.
- Las matrices tienen algunos **atributos añadidos**:
  - `matrix.T` es la matrix transpuesta.
  - `matrix.H` es la matriz compuesta conjugada (compleja)
  - `matrix.I` es la inversa de la matriz (de ser inversible)
  - `matrix.A` es la matriz como `ndarray`

Matrices de NumPy (II)

NumPy **NO** recomienda utilizar la clase `matrix`, ya que dificulta la creación y consistencia de funciones que admitan arrays y matrices. Actualmente, se usan principalmente para interactuar con el módulo `scipy.sparse`, y se espera poder eliminar la clase `matrix` en algún momento del futuro.

Diferencias entre arrays y matrices

```
a = np.arange(1, 5).reshape(2, 2)
m = np.matrix(a)
```

```
print(a)          # [[1 2]
                  #  [3 4]]

a * np.eye(2)     # [[1. 0.]
                  #  [0. 4.]]

a**2              # [[1  4]
                  #  [9 16]]
```

```
print(m)          # [[1 2]
                  #  [3 4]]

m * np.eye(2)     # [[1. 2.]
                  #  [3. 4.]]

m**2              # [[ 7, 10]
                  #  [15, 22]]
```

Más cosas sobre NumPy

- Lo que se ha tratado de NumPy es una parte **mínima** de lo que el módulo ofrece.
  - Puede que sea suficiente para la carrera, pero si se desea explotar todo el potencial que el módulo ofrece, ya sea por necesidad profesional o por curiosidad personal, todavía habría que tratar muchas funcionalidades.
- Por ejemplo, la parte del módulo encargada de las operaciones de álgebra lineal (`numpy.linalg`) incluye algoritmos para **resolver ecuaciones** representadas en forma matricial o tensorial.
- Se recomienda encarecidamente consultar la **página oficial** de NumPy para conocer qué más cosas se pueden hacer con el módulo:
  - **Documentación**: `https://numpy.org/doc/stable/`
  - **Tutoriales**: `https://numpy.org/learn/`

pandas

- pandas es un módulo de PYTHON de código libre, que proporciona utilidades para el **análisis de datos**.
- Centrado en el trabajo con **datos tabulados**.
- Trabaja sobre **NumPy**.
- En su **página web** ofrecen documentación y tutoriales para aprender a utilizar el módulo de forma fácil.

```
import pandas as pd
```

DataFrames de pandas

- La estructura principal con la que se trabaja en pandas es el `DataFrame`.
  - También se utilizan las `Series`, pero son menos importantes.
- Un DataFrame es una estructura de datos etiquetados bidimensional, con **columnas** de tipos potencialmente diferentes.
  - Un DataFrame se puede entender como un diccionario de Series.

```
df = pd.DataFrame(np.arange(16).reshape(4, 4)**2,
    index=list("abcd"), columns=list("ABCD"))
#          a   b   c   d
#    A     0   1   4   9
#    B    16  25  36  49
#    C    64  81 100 121
#    D   144 169 196 225
```

Operaciones con DataFrames: columnas

```
df = pd.DataFrame(np.arange(16).reshape(4, 4)**2,
     index=list("abcd"), columns=list("ABCD"))
```

- **Selecciones** por columnas:
  - El resultado es una Series

```
df["a"]
# A      0
# B     16
# C     64
# D    144
# Name: a, dtype: int32
```

- **Añadir** columna al final:

```
df["sqr"] = df["a"]**2
#       a    b    c    d     sqr
# A     0    1    4    9       0
# B    16   25   36   49     256
# C    64   81  100  121    4096
# D   144  169  196  225   20736
```

- **Añadir** columna al en una posición dada:

```
df.insert(0, 3.1416, df["d"] > 50)
#     3.1416     a    b    c    d
# A   False     0    1    4    9
# B   False    16   25   36   49
# C    True    64   81  100  121
# D    True   144  169  196  225
```

- **Eliminar** columna:

```
del df[3.1416]
```

- Acceso a un **elemento**:

```
df["a"][2]   # 64
```

Operaciones con DataFrames: filas

```
df = pd.DataFrame(np.arange(16).reshape(4, 4)**2,
    index=list("abcd"), columns=list("ABCD"))
```

- **Selecciones** por filas:
  - El resultado es una Series

```
df.loc["A"]
# a    0
# b    1
# c    4
# d    9
# Name: A, dtype: int32
```

- **Selecciones** de varias filas:

```
df[1:3]
#     A    B    C    D
# B  16   25   36   49
# C  64   81  100  121
```

- **Selecciones** por índice de fila:

```
df.iloc[0]
# a    0
# b    1
# c    4
# d    9
# Name: A, dtype: int32
```

- Operaciones de **añadir** fila y acceder a **elementos** análogas a las de columnas.

Operaciones con DataFrames

- Las **operaciones aritméticas entre DataFrames** aplican la operación a los elementos de los DataFrames en la misma localización.
  - En este caso, el resultado tendrá el tamaño del DataFrame más grande, aunque los resultados en los elementos no comunes pueden ser indefinidos.

```
pd.DataFrame(np.arange(4).reshape(2, 2))
    - pd.DataFrame(np.arange(9).reshape(3, 3))
#      0    1   2
# 0  0.0  0.0 NaN
# 1 -1.0 -1.0 NaN
# 2  NaN  NaN NaN
```

- Las **operaciones aritméticas con escalares** aplican la misma operación a todos los elementos del DataFrame, como es de esperar:

```
pd.DataFrame(np.arange(4).reshape(2, 2)) * 3
#    0  1
# 0  0  3
# 1  6  9
```

Operaciones útiles con DataFrames

```
df = pd.DataFrame(np.arange(16).reshape(4, 4)**2,
    columns=list("abcd"), index=list("ABCD"))
```

- **Elemento** en una posición:

```
df.at["B", "b"] # 25
```

- **Elemento** en un **índice**:

```
df.iat[1, 1] # 25
```

- **Media**:

```
df.mean()
# a     56.0
# b     69.0
# c     84.0
# d    101.0
# dtype: float64
```

```
df.mean(axis=1) # 0 = columnas
                # 1 = filas
# A      3.5
# B     31.5
# C     91.5
# D    183.5
# dtype: float64
```

- Operaciones similares a la media:
  - **Varianza**: `df.var()`
  - **Desviación estándar**: `df.std()`
  - **Suma**: `df.sum()`
  - ...

Más cosas sobre pandas

- Al igual que con NumPy, aquí solo se ha hecho una **brevísima introducción** a todo lo que ofrece pandas.
- Se recomienda encarecidamente consultar la **página oficial** de pandas para conocer qué más cosas se pueden hacer con el módulo:
  - **Documentación**: `https://pandas.pydata.org/docs/reference/index.html`
  - **Tutoriales**: `https://pandas.pydata.org/docs/getting_started/index.html`

Gracias por vuestra atención

- Esperamos que el taller os haya resultado útil, y que no se os haya hecho demasiado pesado.
  - Ha sido un taller bastante completo, así que es normal necesitar un tiempo para asimilar todos los conceptos.
- **Repositorio** con el código del taller:
  `https://github.com/0xb01u/TallerPy_HoC2020`
- Si os habéis quedado con **dudas**, o necesitáis cualquier tipo de **ayuda**, contactadnos en **Telegram**:
  - @bomilk
  - @jcgd2415
- Y ahora, **¡a programar!**

```
exit()
```