GUL
UC3M

uc3m | Universidad Carlos III de Madrid

Python para pythipiantes

Por Luis Daniel Casais Mezquida

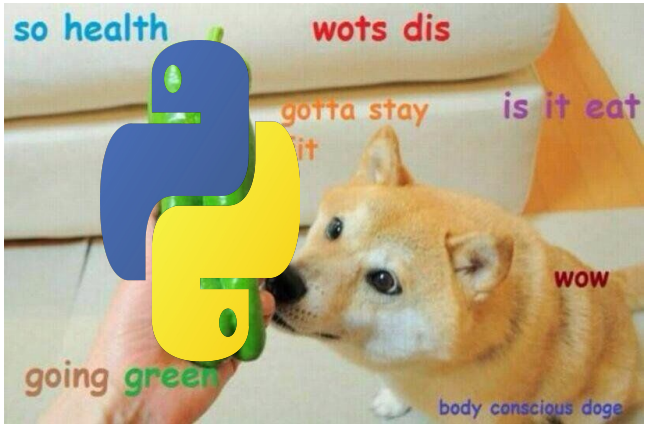
luisdaniel.casais@alumnos.uc3m.es
gul.es | @guluc3m

Noviembre 2021

Transparencias disponibles en cloud.gul.es/ftp

Tabla de contenidos

¿Qué es Python?



- Lenguaje de programación* open-source (yay!)
- Enfocado en la claridad y legibilidad del código
- Multiusos, aunque no muy eficiente
- Orientado a objetos
- **Interpretado****, no compilado
- Dinámico y no tipado
- *Garbage-collected*
- La **indentación** y el **espaciado** importan, y mucho

En definitiva, un lenguaje muy flexible y **accesible** para principiantes.

Pero, ¡cuidado! Es muy fácil hacer *spaghetti code*.

¿Qué es Python?

- Los mensajes de error son mejores que en C/C++

¿Qué es Python?

Versiones

Python cuenta con dos versiones distintas:

- **3.x** → Sigue actualizándose, actualmente está en 3.10.0
- **2.x** → Deprecado desde 2020, pero todavía usado

Ambas versiones cuentan con **distinta sintaxis**, y **no son compatibles** entre sí.

Se puede traducir de Python 2 a 3 usando 2to3.

En este taller nos centraremos en **Python 3**.

¿Qué es Python?

Instalación

Windows/Mac

1. Descargar la última versión desde python.org/downloads/
2. Ejecutar el .exe y añadir Python al PATH

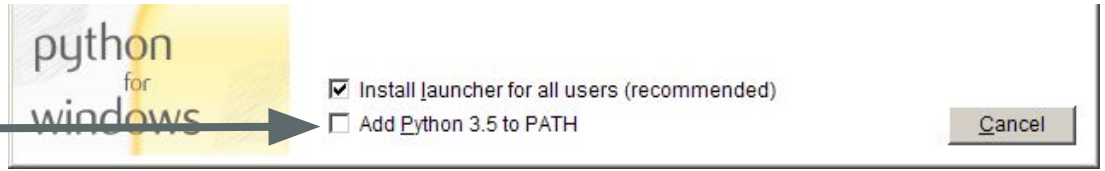
Linux

```
$ sudo apt-get install python3
```

Puedes comprobar la versión con:

```
$ python3 --version
```

CLICK!



¿Qué es Python?

Ejecución

Al ser un lenguaje interpretado, Python se puede ejecutar usando su **terminal** o a través de *scripts* .py.

Abrir la terminal de Python: `$ python3`

```
ldcas@LU15D4-B4BY:/mnt/c/Users/ldcas$ python3
Python 3.8.10 (default, Sep 28 2021, 16:10:42)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Ejecutar un *script*: `$ python3 script.py`

Es recomendable trabajar tanto con *scripts* como con la terminal a la hora de programar, para probar cosas y *debuggear*.

Sintaxis

- Cada instrucción termina con un *endline* (ni “ *;* ” ni leches)
- Los bloques de código van marcados por la **indentación**. Se recomienda usar 4 espacios (equivalente a un *tab*, pero **nunca mezcles**)
- *Case-sensitive* y *space-sensitive* (excepto operadores)
- Se pueden usar `"` y `'` para las mismas cosas, pero no son intercalables entre sí
- Se utiliza `\` como secuencia de escape
- Comentario de línea: `# python mola`
- Comentario de bloque:

```
"""
python
mola
"""
```

Sintaxis

Variables

- Son **punteros** a objetos
- Las variables tienen que ser **declaradas e inicializadas en la misma línea**
- Son **reutilizables**, no dependen de ningún tipo específico
- Se pueden declarar más de una en la misma línea
- Se pueden asignar una variable a otra → se **copia** el objeto
- No hay variables constantes. Por convenio, se usan nombres en MAYÚSCULAS

```
a = 0                          # asignación estándar
a, b = "caca"                  # las dos se asignan a lo mismo
a, b, c = 1, False, 1 + 2j     # se asignan por orden
b = a                          # copia el valor
CONSTANTE = 69                 # no se debería cambiar
(convenio)
```

Sintaxis

Variables

- El *scope* de una variable es la función que la contiene (*local scope*)
- El resto de variables son **globales**
- La *keyword* `global` hace una variable (dentro de una función) global
- Para **modificar** una variable global dentro de una función hay que usar `global` (aunque no es una buena práctica)

```
a = 0               # variable global
def func():
    b = 1           # variable local
    global a, c     # crea global c y accede a global a
    c = 2
    a = 3           # modifica a
c = 4               # modifica c
```

Sintaxis

Tipos de datos y *castings*

Son todos **objetos** predefinidos.

- `int`: No tienen límite (RAM)
- `float`: *single/double precision*
- `complex`: `a+bj`
- `bool`: `True` / `False`
- `str` : Caracteres ASCII. `"abc"` ≡ `'abc'`
- `None`: Sin valor (objeto vacío)

```
a = 69        # int
b = 4.20      # float
c = 4 + 20j   # complex
d = False     # bool
e = "caca"    # str
```

Ints, floats, bools y strings son inmutables, es decir, no se pueden modificar.

Sintaxis

Tipos de datos y *castings*

- Los tipos de datos se convierten **automáticamente**.
- Para convertir manualmente (*casting*) se llama al constructor del tipo de dato.

```
>>> a = 69
>>> b = str(a)   # casting int → str
>>> b
'69'
```

Sintaxis

Tipos de datos y *castings*

¡Cuidado con *castings* de strings al resto de tipos!

- Sólo funciona si son números. Ej. `"69"`
- Si es una string vacía, sólo se puede convertir a Bool

```
>>> a = "caca"
>>> int(a)
ValueError: invalid literal for int() with base 10: 'caca'
```

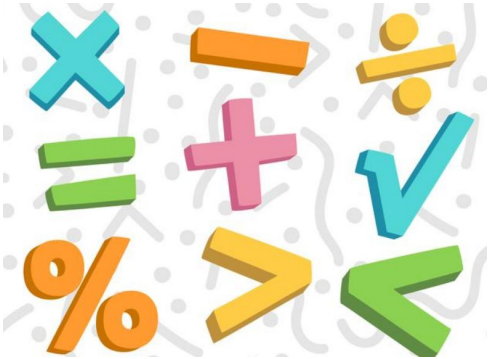
Para castings hacia boolean, `0` es `False` y todo lo demás es `True`.

- `None` cuenta como `0`
- Una string vacía (`""`) cuenta como `0`

```
>>> bool(None and "")
False
```

Sintaxis

Operadores



- Asignación: `=`
- Aritméticos: `+`, `-`, `*`, `/`
- Divisores: `%` (resto), `//` (división entera)
- Potencias: `**`

Estos operadores se pueden combinar con la asignación (`+=`, `-=`, ...).

Los siguientes retornan Boolean:

- Relacionales: `==`, `<`, `>`, `<=`, `>=`, `!=`
- Lógicos: `and`, `or`, `not`
- Evaluación : `is`, `is not`. Evalúa si son **se refieren al mismo objeto** (diferente a `==`)
- Pertenencia : `in`, `not in`. Evalúa si el elemento pertenecen a la variable

Sintaxis

Operadores

Orden de precedencia:

1. `()`
2. `**`
3. +, – (como signo)
4. `*`, `/`, `%`, `//`
5. +, – (como operador)
6. `<`, `>`, `<=`, `>=`
7. `==`, `!=`
8. `=`
9. `is`, `is not`
10. `in`, `not in`
11. `and`, `or`, `not`

```
>>> a = 1 + 2 * 3**2
>>> a
19
>>> a = (1 + 2) * 3**2
>>> a
27
```

Ante la duda, ~~la más te~~ usa paréntesis.

Sintaxis

Operadores

Strings y listas tienen ciertos operadores específicos:

- `+` concatena y `*` repite *n* veces (debe ser un Entero)
- `var[n]` **accede** al elemento con índice *n*
- `var[-n]` ≡ `var[len(var)-n]`
- `var[n:m]` crea una *substring* de *n* a *m*-1
- `var[n:]` ≡ `var[n:len(var)]`
- `var[:n]` ≡ `var[0:n]`
- `elem in var` evalúa si *elem* es un elemento de *var*

```
>>> a = "@guluc3m"
>>> a[8]
IndexError: string index
out of range
>>> a[0]
'@'
>>> a[-1]
'3'
>>> a[1:4]
'gul'
>>> a[4:]
'uc3m'
>>> "u" in var
True
```

Sintaxis

Estructuras de datos

Lista - `list`

Secuencia de elementos (*linked list*):

- **Heterogénea**: permite elementos de distintos tipos
- Finita y **mutable** (ampliable)
- Ordenada e **indexada**

El **constructor** es `[elem1,elem2,<...>]`.

- Los elementos son modificables
- Se puede hacer *casting* de string a lista, y viceversa

```
>>> a = [0, True, "xd"]
>>> str(a)
"[0, True, "xd"]"
>>> b = "caca"
>>> c = list(b)
['c', 'a', 'c', 'a']
>>> str(c)
"['c', 'a', 'c', 'a']"
```

Sintaxis

Estructuras de datos

Lista - `list`

Al ser un objeto, cuenta con métodos asociados:

- `var.append(elem)`: Añade *elem* al final de *var*
- `var.pop()`: Elimina el último elemento y lo **retorna**
- `var.pop(pos)`: Elimina el elemento en *pos* y lo **retorna**
- `var.insert(pos,elem)`: Añade *elem* a *var* en *pos*
- `foo = var.copy()`: **Copia** la lista *var* a *foo*.
- `foo = var` copia el objeto, con el puntero al *head* de la lista por lo que **los cambios en *var* afectan a *foo***

```
>>> a = [0, 1]
>>> b = a  # no hacer con listas
>>> a[0] = 1
>>> b
[1, 1]
```

Sintaxis

Estructuras de datos

Tuple - `tuple`

Listas inmutables.

El constructor es `(elem1,elem2,<...>)`.

- Los elementos **no** son modificables
- Se puede hacer *casting* a string o una lista, y viceversa
- Más eficientes en memoria

Útiles para guardar información constante.

```
>>> a = (0, True, "xd")
>>> b = list(a)
[0, True, "xd"]
>>> b[0] = 1
>>> b
[1, True, "xd"]
```

Sintaxis

Estructuras de datos

Lo siguiente afecta tanto a listas como a tuples:

- Como una string, pueden estar vacíos.
- ***Unpacking*:** Guardar los elementos en varias variables a la vez. El número de variables tiene que ser el mismo que el de elementos.

```
>>> myList = [0, True, "xd"]
>>> a, b, c = myList
>>> print(a,b,c)
0 True xd
```

***Nesting*:** Estructuras dentro de estructuras. Útiles para matrices. Para acceder a un elemento se usa un operador `[]` por cada "nivel". Son difíciles de copiar, `copy.deepcopy(var)` (módulo copy)

```
>>> matrix = ([0,1],[2,3])
>>> matrix[1][0]
2
```

Sintaxis

Estructuras de datos

Diccionario - `dict`

Secuencia de **pares** *key - element*:

- **Heterogénea**. La *key* debe ser **inmutable** y única para la estructura
- Finita y **mutable** (ampliable)
- I**ndexada** por la *key*

El constructor es `{key1:elem1,<...>}`.

"*Son básicamente JSONs*" — mi profe.

Sintaxis

Estructuras de datos

Diccionario - `dict`

- `foo = var.get(key)`: Accede a un elemento. En caso de no existir, retorna `None`
- `var[key] = elem`: Modifica un elemento. Si *key* no existe, crea uno nuevo.
- `var.pop(key)`: Borra un elemento
- `foo = var.copy()`: Copia el diccionario

Al hacer *casting* a una lista/tuple, sólo se pasan las *keys*.

```
>>> a =
{"caca":1,True:[0,2]}
>>> a.get(True)[0]
0
>>> a[2] = 3
>>> a.pop("caca")
1
>>> a
{True: [0, 2], 2: 3}
>>> a[2] = "ey"
>>> a
```

Sintaxis

Control de flujo

If - `if condition:`

- *condition* debe retornar un Boolean

Else - `else:`

- Debe ir siempre después de un If (ó Elif)

Elif - `elif condition:`

- Else + If, debe de ir después de un If

También existe un *switch*: `match pattern:` (Python 3.10+)

```
if(a == 69):
    b = "nice"
elif(a in sumStuff):
    b = True
else:
    b = False
```

Sintaxis

Bucles

While - `while(condition):`

- Igual que en C/C++, Java, ...
- El bloque se repite mientras *condition* retorne True
- Se comprueba **al principio** de la iteración
- Encima táctico

```
>>> a = 0
>>> while(a <= 5):
...     a += 1
>>> a
6
```

Sintaxis

Bucles

For - `for elem in object:`

- Son equivalentes a bucles ***for-each***
- Recorre los elementos de *object*
- *object* debe de ser un objeto **iterable**
- Se asigna a *elem* cada iteración
- Cuando llega al final de *object*, termina

```
>>> a = (1, True, "a")
>>> b = []
>>> i = 0
>>> for elem in a:
...     b[i] = elem
...     i += 1
>>> b
[1, True, "a"]
```

Se puede implementar un bucle *for* clásico con un bucle While o usando `for i in range(max)`.

Sintaxis

Bucles

Se pueden usar **bifurcaciones** para modificar el comportamiento de los bucles:

`break`

- Sale del bucle
- Nunca mira atrás

`continue`

- Termina la iteración actual del bucle
- Empieza la siguiente iteración

```
for elem in myList:
    if elem == 69:
        break
    elif elem == 420:
        continue
    elem += 1
```

Se usa `pass` como NOP (*No OPeration*), equivale a no escribir nada.

Sintaxis

Funciones

`def function(param1, ...):`

- Se utiliza *pass by value* para *datatypes* y *pass by reference* para estructuras y objetos
- No hay *overloading*
- Deben de ser definidas antes de ser usadas
- Para parámetros con valor por defecto se usa `param = value`
- Para retornar un valor/variable se usa `return var`
- Se pueden retornar varios valores → `return var1, var2` → retorna un Tuple

Se llaman con `function(param1, ...)`

```
def divide(a, b = 1):
    c = a//b
    r = a%b
    return c, r

coc, res = divide(3)
```

Sintaxis

Funciones

Python cuenta con ciertas funciones predefinidas:

- `type(var)`: Retorna el tipo (o clase) de una variable
- `isinstance(var,type)`: Dice si *var* pertenece al tipo *type*. Retorna boolean
- `len(object)`: Retorna el tamaño (**no** en bytes) del objeto
- `range(a,b,step)`: Genera números **enteros** en el rango [*a*, *b*), con intervalo *step*. Los parámetros *b* y *step* son opcionales
- `print()`: Imprime en pantalla. Toma Strings o variables como argumentos, y las concatena.
- `input()`: Como print, pero lee el input del teclado y lo retorna

Programación Orientada a Objetos (OOP)

- Todos los objetos pertenecen a una **clase**
- Se declara una clase con `class Name:`
- Para crear un objeto se usa `Name()`
- Al crear un objeto se **ejecuta** el código que contiene la clase
- Se pueden asignar a variables con `var = Name()`
- No hay límite de objetos creados (**instancias**) de una clase
- Se usa `self` para referirse al propio objeto
- Soportan **composición** y **herencia**

```
>>> a = 0
>>> class Name:
...     global a
...     a = 1
...
>>> Name()
<__main__.Name object at
0x7fbd0e7eec70>
>>> a
1
```

Programación Orientada a Objetos (OOP)

Atributos y Métodos

Los **atributos** son variables locales del objeto.

- Son públicos por defecto
- Se accede con `var.atr`, siendo `var` una variable conteniendo el objeto y *atr* el atributo (`self.atr` dentro del objeto)
- Se pueden hacer privados llamándolos `__atr`

Los **métodos** son funciones locales del objeto.

- Públicos excepto si se usa `def __method():`
- Si se usan atributos hay que pasar `self` como parámetro

```
class Ganso:
    tamaño = 69
    __comida = 1

    def comer(self):
        self.__comida += 1

    def graznar():
        print("Honk")
```

Programación Orientada a Objetos (OOP)

Atributos y Métodos

Existen métodos especiales con funcionalidades especiales, con formato `__meth__()`:

Constructor - `def __init__(self):`

- Se ejecuta nada más crear el objeto
- Útil para iniciar atributos con parámetros y comprobarlos

Str - `def __str__(self):`

- Retorna la string que escribir cuando se *printea* un objeto de la clase

```
class Ganso:
    def __init__(self, tam=69):
        if(tam > 0):
            self.tamaño = tam
        else:
            self.tamaño = 1
        self.__comida = 1

    def __str__(self):
        return str(self.tamaño)
         +",
"+str(self.__comida)

    def __repr__(self):
        return self.__str__()
```

Programación Orientada a Objetos (OOP)

Propiedades

Se usan para **encapsular** atributos, prevenir que tomen ciertos valores y poner valores por defecto (similar a *getter/setter*).

- *Getter*:

```
@property
def atr(self):
    return self.__atr
```

- *Setter*:

```
@atr.setter
def atr(self,atr):
    [comprobación]
    self.__atr = atr
```

Programación Orientada a Objetos (OOP)

Propiedades

```
class Ganso:
    def __init__(self,tam=69):
        if(tam > 0):
            self.tamaño = tam
        else:
            self.tamaño = 1
    @property
    def tamaño(self):
        return self.__tamaño
    @tamaño.setter
    def tamaño(self,tamaño):
        if(tamaño > 0):
            self.__tamaño = tamaño
        else:
            self.__tamaño = 1
```

```
>>> a = Ganso()
>>> a.tamaño
69
>>> b = Ganso(3)
>>> b.tamaño
3
>>> b.tamaño = 420
>>> b.tamaño
420
>>> b.tamaño = -1
>>> b.tamaño
1
```

Programación Orientada a Objetos (OOP)

Herencia

Usado para **expandir** y particularizar una clase.

- Todos los métodos y atributos (y propiedades) de la clase madre los tiene la clase hijo
- Se accede a los métodos de la madre con `super()`, y se pueden **sobrescribir**
- Se crean con `class Hijo(Madre):`
- Un objeto de la clase hijo también lo es de la clase madre (**polimorfismo**)

```
class GansoRobot(Ganso):
    def __init__(self,tam=69,poder):
        super().__init__(tam)
        self.poder = poder

    def graznar():  # sobreescribe
        print("[Metallic] Honk")

    def atacar(self):
        if(self.poder > 0):
            print("Toma eso")
            self.poder -= 1
```

Hello $user

script.py

```
class App():
    """
    Hello $user
    """
    name = input("Enter your name: ")
    print("Hello", name)

if __name__ == '__main__':  # barrera
    App()
```

```
$ python3 script.py
Enter your name: Luisda
Hello Luisda
```

Módulos

Los *scripts* de Python se pueden **importar** en otros scripts, y se llaman **módulos**.

Se usa `import módulo`.

- Se importan variables, funciones, y objetos
- Por defecto coge las variables del entorno
- Para importar clases específicas se usa `from módulo import Clase`
- También se usa para especificar el *path* de un *script*: `from path import script`

El **Package Installer for Python** (pip) **Instala** módulos de Python en el *environment*, para que luego puedan ser importados.

- Se usa `$ pip install módulo`
- Incluído a partir de python 3.4, recomendable añadir al PATH en Windows

Virtual Environment (venv)

El venv es un **entorno** donde instalar las dependencias necesarias para un proyecto de Python.

- Todas las dependencias (módulos) se guardan en la carpeta del venv
- Evita tener que instalar módulos directamente en el sistema
- Hace los proyectos portables y **fáciles de replicar**

Se instala con `$ sudo pip install virtualenv`

Recomendable usar siempre que se use un git en remoto (GitHub, GitLab…), o en general para cualquier proyecto que vaya a ser usado por más personas.

Virtual Environment (venv)

Creación y uso

Se crea el venv con `$ python3 -m virtualenv *ruta*`.

- Es recomendable que la ruta del venv sea la una subcarpeta del proyecto, es decir, `./venv`.

Se activa con `$ source *ruta*/bin/activate` para Linux y
`> *ruta*\Script\activate.bat` para Windows.

```
ldcas@LU15D4-B4BY:/mnt/c/Users/ldcas/tmp$ source ./venv/bin/activate
(venv) ldcas@LU15D4-B4BY:/mnt/c/Users/ldcas/tmp$
```

Si se usa git, la carpeta debería estar en el .gitignore.

Virtual Environment (venv)

Venv y pip

Al instalar módulos con pip dentro del venv, se instalan en la subcarpeta y no en el sistema local.

Para facilitar el instalar módulos en otro sistema, se usa el fichero requirements.txt.

- Se crea con `$ pip freeze > requirements.txt`
- Se instalan dependencias con `$ pip install -r requirements.txt`

Información extra

Info más detallada sobre Python 3:

- Documentación oficial: docs.python.org/3/
- La mítica: w3schools.com/python/

Integrated Development Environments (IDEs):

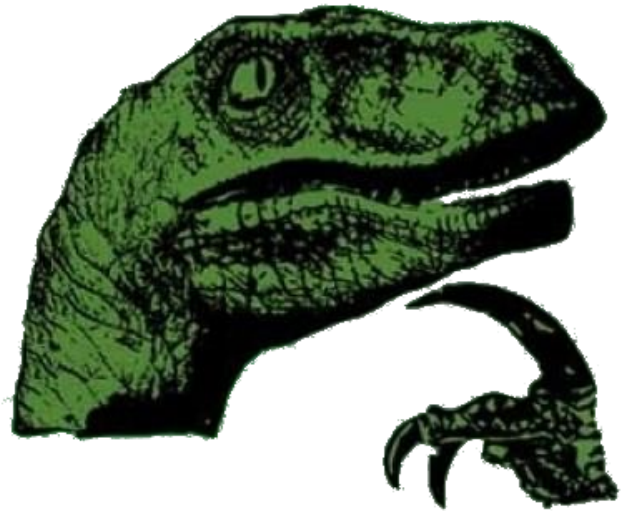
- PyCharm: Integra venv y más
- Spyder (Anaconda)
- VSCode con la extensión de Python

Guía de estilo (PEP8): python.org/dev/peps/pep-0008/

Módulos que controlar: os, unittest, re, random, math, exception

Repositorio con ejemplos autodidácticos: github.com/Patataman/PythonBasic

¿Preguntas, reclamos, improperios?



fin

luisdaniel.casais@alumnos.uc3m.es
gul.es | @guluc3m

Transparencias disponibles en cloud.gul.es/ftp