Sistemas Operativos en Red  
UD10 - Extra - Introducción a ShellScripting con Python 3

short line

Autor: Sergi García y material con licencia CC BY SA de <https://learnxinyminutes.com/docs/es-es/python-es/>

Actualizado Enero 2025

Licencia

**Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA)**: No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

**📖 Importante**

**❕ Atención**

**💬 Interesante**

**Índice**

[**1. Introducción 2**](#_g7i82c35p6k)

[**2. Resumen características Python 3 2**](#_pcye9v7uoz0h)

[**3. Elementos del lenguaje Python 3 3**](#_1b17ag3uvbih)

[**3.1 Comentarios 3**](#_87dljvcql42g)

[**3.2 Tipos de datos y operadores 3**](#_d58j7lrmz6a0)

[**3.3 Variables y colecciones 6**](#_rfbo3ybj2vpa)

[**3.4 Control de flujo 10**](#_ahdqajdbvf5b)

[**3.5 Funciones 13**](#_wcxxdvqdy9qt)

[**3.6 Clases 15**](#_i4mnnyx6m2x5)

[**3.7 Módulos 16**](#_bfvl3lx14x8t)

[**3.8 Avanzado: generadores y decoradores 17**](#_me7pima78wx6)

[**4. Usar Python para hacer Scripts de Sistemas Operativos 18**](#_fvy21qn1ft33)

[**5. Bibliografía 20**](#_9inb55ixvere)

Unidad 10. Introducción a ShellScripting con Python 3

# Introducción

En este documento, vamos a realizar un resumen de los principales elementos del lenguaje Python, basándonos en la documentación con licencia CC BY SA de <https://learnxinyminutes.com/docs/es-es/python-es/>, comentando cada elemento y en algunos casos añadiendo ejemplos adicionales.

# Resumen características Python 3

Por lo que respecta Python, comentar que se trata de una opción muy atractiva para comenzar a codificar o para aprender rápidamente a programadores expertos. Algunas características:

* **Sintaxis sencilla**: los programas escritos con Python son auto-expresivos, muy cercanos a un algoritmo escrito en pseudocódigo o lenguaje natural.
* **Muy potente**: en pocas líneas de código, Python puede ejecutar muchas acciones (y habitualmente implementadas de una forma muy óptima) que con otros lenguajes de programación equivaldrían a muchas líneas más para poder conseguir el mismo efecto.
* **Lenguaje interpretad**o: las instrucciones son traducidas y ejecutadas instrucción a instrucción. No hay ficheros de código intermedio, ni tampoco tiempo de compilación.
* **Lenguaje sin obligación de declarar tipos de datos**: este aspecto puede considerarse una ventaja o no por al desarrollador de código.
* **Curva de aprendizaje suave**: para comenzar a programar desde cero, Python es de las mejores opciones. Además, para programadores expertos es un lenguaje sencillo de aprender.

# Elementos del lenguaje Python 3

## Comentarios

Los comentarios se realizan con el carácter “#” para una línea y “tres comillas” para multilínea. Además, las 3 comillas pueden utilizarse para definir cadenas multilínea.

| *# Comentarios de una línea comienzan con una almohadilla (o signo gato)*  """ Strings multilinea pueden escribirse  usando tres "'s, y comunmente son usados  como comentarios. """ |
| --- |

## Tipos de datos y operadores

| *####################################################* *## 1. Tipos de datos primitivos y operadores.* *####################################################* |
| --- |

La nomenclatura para esta sección es “operación # => resultado esperado”, donde la parte a la derecha del carácter “#” es un comentario y solo nos da información de como va a funcionar la operación.

Si en Python pones un número, obtienes simplemente ese número.

| *# Tienes números* 3 *# => 3* |
| --- |

Si realizas operaciones aritméticas con enteros, obtienes el resultado con un número entero. Los paréntesis modifican la precedencia entre operadores.

| *# Matemática es lo que esperarías 1 + 1 # => 2 8 - 1 # => 7 10 \* 2 # => 20 # Refuerza la precedencia con paréntesis (1 + 3) \* 2 # => 8* |
| --- |

La división, aunque entre enteros, devuelve un tipo de dato “float” (decimal) si se hace con “/”, pero si se desea un resultado entero (con truncado de decimales) puedes utilizar “//”.

| *# Excepto la división la cual por defecto retorna un número 'float' (número de coma flotante)* 35 / 5 *# => 7.0* *# Sin embargo también tienes disponible división entera* 34 // 5 *# => 6* |
| --- |

Si en una operación aritmética, alguno de los dos operadores es un “float”, el resultado siempre es un float (se convierte al tipo de datos que mayor engloba).

| *# Cuando usas un float, los resultados son floats* 3 \* 2.0 *# => 6.0* |
| --- |

Aquí vemos el tipo de datos “boolean” y los operadores lógicos que nos devuelven un “boolean”.

| *# Valores 'boolean' (booleanos) son tipos primitivos* **True** **False**  *# Niega con 'not'* **not** **True** *# => False* **not** **False** *# => True*   *# Igualdad es ==* 1 == 1 *# => True* 2 == 1 *# => False*  *# Desigualdad es !=* 1 != 1 *# => False* 2 != 1 *# => True*  *# Más comparaciones* 1 < 10 *# => True* 1 > 10 *# => False* 2 <= 2 *# => True* 2 >= 2 *# => True*  *# ¡Las comparaciones pueden ser concatenadas!* 1 < 2 < 3 *# => True* 2 < 3 < 2 *# => False* |
| --- |

Aquí observamos como definir “Strings” (cadenas de caracteres) y como operar con ellos (formato, concatenación, acceso a un elemento, etc.)

| *# Strings se crean con " o '* "Esto es un string." 'Esto también es un string'  *# ¡Strings también pueden ser sumados!* "Hola " + "mundo!" *# => "Hola mundo!"*  *# Un string puede ser tratado como una lista de caracteres* "Esto es un string"[0]  *'E'*  *# .format puede ser usaro para darle formato a los strings, así:* "{} pueden ser {}".format("strings", "interpolados")  *# Puedes reutilizar los argumentos de formato si estos se repiten.* "{0} sé ligero, {0} sé rápido, {0} brinca sobre la {1}".format("Jack", "vela") *# => "Jack sé ligero, Jack sé rápido, Jack brinca sobre la vela"* *# Puedes usar palabras claves si no quieres contar.* "{nombre} quiere comer {comida}".format(nombre="Bob", comida="lasaña") *# => "Bob quiere comer lasaña"* *# También puedes interpolar cadenas usando variables en el contexto* nombre = 'Bob' comida = 'Lasaña' f'{nombre} quiere comer {comida}' *# => "Bob quiere comer lasaña"* |
| --- |

None es un objeto predefinido en Python, utilizado para comparar si algo es “nada”.

| *# None es un objeto* **None** *# => None*  *# No uses el símbolo de igualdad `==` para comparar objetos con None* *# Usa `is` en su lugar* "etc" **is** **None** *# => False* **None** **is** **None** *# => True*  *# None, 0, y strings/listas/diccionarios/conjuntos vacíos(as) todos se evalúan como False.* *# Todos los otros valores son True* bool(0) *# => False* bool("") *# => False* bool([]) *# => False* bool({}) *# => False* bool(set()) *# => False* |
| --- |

## Variables y colecciones

| *####################################################* *## 2. Variables y Colecciones* *####################################################* |
| --- |

La función “print” nos permite imprimir cadenas de caracteres.

| *# Python tiene una función para imprimir* print("Soy Python. Encantado de conocerte") |
| --- |

En Python no es necesario declarar variables antes de utilizarlas. Una convención es usar \_ para separar las palabras, pero hay otras como Camel Case <https://es.wikipedia.org/wiki/Camel_case>

| *# No hay necesidad de declarar las variables antes de asignarlas. una\_variable = 5 # La convención es usar guiones\_bajos\_con\_minúsculas una\_variable # => 5 otraVariable = 3 # Aqui en formato Camel Case otraVariable # => 3 # Acceder a variables no asignadas previamente es una excepción. # Ve Control de Flujo para aprender más sobre el manejo de excepciones. otra\_variable # Levanta un error de nombre* |
| --- |

La principal colección de elementos en Python son las listas. Aquí vemos ejemplos de uso:

| *# Listas almacena secuencias* lista = [] *# Puedes empezar con una lista prellenada* otra\_lista = [4, 5, 6]  *# Añadir cosas al final de una lista con 'append'* lista.append(1) *#lista ahora es [1]* lista.append(2) *#lista ahora es [1, 2]* lista.append(4) *#lista ahora es [1, 2, 4]* lista.append(3) *#lista ahora es [1, 2, 4, 3]* *# Remueve del final de la lista con 'pop'* lista.pop() *# => 3 y lista ahora es [1, 2, 4]* *# Pongámoslo de vuelta* lista.append(3) *# Nuevamente lista ahora es [1, 2, 4, 3].* |
| --- |

Para acceder a elementos de una lista, accedemos como accederemos en otros lenguajes a un array: si tiene N elementos, con valores del 0 al N-1. Además Python permite referencia negativas. Por ejemplo, -1 en una lista de N elementos, equivale a acceder al elemento “N-1”.

| *# Accede a una lista como lo harías con cualquier arreglo* lista[0] *# => 1* *# Mira el último elemento* lista[-1] *# => 3* *# Mirar fuera de los límites es un error 'IndexError'* lista[4] *# Levanta la excepción IndexError* |
| --- |

Las listas permite obtener una nueva lista formada por un rango de elementos usando “:”. La parte izquierda al “:” es donde comienza, y la parte derecha donde acaba. Si se mete un segundo “:”, indica el número de pasos del rango a tomar. Al final sigue una sintaxis “lista[inicio:final:pasos]”.

A continuación, algunos ejemplos de rangos y otras operaciones (concatenación, comprobar elementos, tamaño, borrado, etc.) con listas:

| *# Puedes mirar por rango con la sintáxis de trozo.* *# (Es un rango cerrado/abierto para los matemáticos.)* lista[1:3] *# => [2, 4]* *# Omite el inicio* lista[2:] *# => [4, 3]* *# Omite el final* lista[:3] *# => [1, 2, 4]* *# Selecciona cada dos elementos* lista[::2] *# =>[1, 4]* *# Invierte la lista* lista[::-1] *# => [3, 4, 2, 1]* *# Usa cualquier combinación de estos para crear trozos avanzados* *# lista[inicio:final:pasos]*  *# Remueve elementos arbitrarios de una lista con 'del'* **del** lista[2] *# lista ahora es [1, 2, 3]*  *# Puedes sumar listas* lista + otra\_lista *# => [1, 2, 3, 4, 5, 6] - Nota: lista y otra\_lista no se tocan*  *# Concatenar listas con 'extend'* lista.extend(otra\_lista) *# lista ahora es [1, 2, 3, 4, 5, 6]*  *# Verifica la existencia en una lista con 'in'* 1 **in** lista *# => True*  *# Examina el largo de una lista con 'len'* len(lista) *# => 6* |
| --- |

Otro elemento (menos utilizado en Python que las listas) son las tuplas. Las tuplas son como las listas, solo que son inmutables (no podemos cambiar valores, añadir, borrar, etc.).

| *# Tuplas son como listas pero son inmutables.* tupla = (1, 2, 3) tupla[0] *# => 1* tupla[0] = 3 *# Levanta un error TypeError*  *# También puedes hacer todas esas cosas que haces con listas* len(tupla) *# => 3* tupla + (4, 5, 6) *# => (1, 2, 3, 4, 5, 6)* tupla[:2] *# => (1, 2)* 2 **in** tupla *# => True*  *# Puedes desempacar tuplas (o listas) en variables* a, b, c = (1, 2, 3) *# a ahora es 1, b ahora es 2 y c ahora es 3* *# Tuplas son creadas por defecto si omites los paréntesis* d, e, f = 4, 5, 6 *# Ahora mira que fácil es intercambiar dos valores* e, d = d, e *# d ahora es 5 y e ahora es 4* |
| --- |

Otra estructura de datos interesante y óptima es la implementación de diccionarios (es decir, asociación clave/valor) en Python mediante la estructura “{ }”. A continuación vemos algunos ejemplos.

| *# Diccionarios relacionan claves y valores* dicc\_vacio = {} *# Aquí está un diccionario pre-rellenado* dicc\_lleno = {"uno": 1, "dos": 2, "tres": 3}  *# Busca valores con []* dicc\_lleno["uno"] *# => 1*  *# Obtén todas las claves como una lista con 'keys()'. Necesitamos envolver la llamada en 'list()' porque obtenemos un iterable. Hablaremos de eso luego.* list(dicc\_lleno.keys()) *# => ["tres", "dos", "uno"]* *# Nota - El orden de las claves del diccionario no está garantizada.* *# Tus resultados podrían no ser los mismos del ejemplo.*  *# Obtén todos los valores como una lista. Nuevamente necesitamos envolverlas en una lista para sacarlas del iterable.* list(dicc\_lleno.values()) *# => [3, 2, 1]* *# Nota - Lo mismo que con las claves, no se garantiza el orden.*  *# Verifica la existencia de una llave en el diccionario con 'in'* "uno" **in** dicc\_lleno *# => True* 1 **in** dicc\_lleno *# => False*  *# Buscar una llave inexistente deriva en KeyError* dicc\_lleno["cuatro"] *# KeyError*  *# Usa el método 'get' para evitar la excepción KeyError* dicc\_lleno.get("uno") *# => 1* dicc\_lleno.get("cuatro") *# => None* *# El método 'get' soporta un argumento por defecto cuando el valor no existe.* dicc\_lleno.get("uno", 4) *# => 1* dicc\_lleno.get("cuatro", 4) *# => 4*  *# El método 'setdefault' inserta en un diccionario solo si la llave no está presente* dicc\_lleno.setdefault("cinco", 5) *#dicc\_lleno["cinco"] es puesto con valor 5* dicc\_lleno.setdefault("cinco", 6) *#dicc\_lleno["cinco"] todavía es 5* *# Elimina claves de un diccionario con 'del'* **del** dicc\_lleno['uno'] *# Remueve la llave 'uno' de dicc\_lleno* |
| --- |

Otra estructura de datos óptima para este proceso son los conjuntos. Permite hacer de manera óptima operaciones relacionadas con los conjuntos (intersección, unión, etc.).

| *# Sets (conjuntos) almacenan conjuntos* conjunto\_vacio = set() *# Inicializar un conjunto con montón de valores. Yeah, se ve un poco como un diccionario. Lo siento.* un\_conjunto = {1,2,2,3,4} *# un\_conjunto ahora es {1, 2, 3, 4}*  *# Añade más valores a un conjunto* conjunto\_lleno.add(5) *# conjunto\_lleno ahora es {1, 2, 3, 4, 5}*  *# Haz intersección de conjuntos con &* otro\_conjunto = {3, 4, 5, 6} conjunto\_lleno & otro\_conjunto *# => {3, 4, 5}*  *# Haz unión de conjuntos con |* conjunto\_lleno | otro\_conjunto *# => {1, 2, 3, 4, 5, 6}*  *# Haz diferencia de conjuntos con -* {1,2,3,4} - {2,3,5} *# => {1, 4}*  *# Verifica la existencia en un conjunto con 'in'* 2 **in** conjunto\_lleno *# => True* 10 **in** conjunto\_lleno *# => False* |
| --- |

## Control de flujo

| *####################################################* *## 3. Control de Flujo* *####################################################* |
| --- |

Aquí vemos ejemplos de como utilizar la estructura de control de flujo “if”:

| *# Creemos una variable para experimentar* some\_var = 5  *# Aquí está una declaración de un 'if'. ¡La indentación es significativa en Python!* *# imprime "una\_variable es menor que 10"* **if** una\_variable > 10:  print("una\_variable es completamente mas grande que 10.") **elif** una\_variable < 10: *# Este condición 'elif' es opcional.*  print("una\_variable es mas chica que 10.") **else**: *# Esto también es opcional.*  print("una\_variable es de hecho 10.") |
| --- |

Aquí vemos como utilizar la estructura “for” para iterar sobre cada elemento de los elementos que Python considera “iterables” (listas, tuplas, diccionarios, etc.).

| """ For itera sobre iterables (listas, cadenas, diccionarios, tuplas, generadores...) imprime:  perro es un mamifero  gato es un mamifero  raton es un mamifero """ **for** animal **in** ["perro", "gato", "raton"]:  print("{} es un mamifero".format(animal)) |
| --- |

La función “range” es un generador de números. Nos puede ayudar para realizar iteraciones numéricas utilizando for:

| """ `range(número)` retorna un generador de números desde cero hasta el número dado imprime:  0  1  2  3 """ **for** i **in** range(4):  print(i) |
| --- |

La estructura de control de flujo “While”, itera mientras una condición sea cierta.

| """ While itera hasta que una condición no se cumple. imprime:  0  1  2  3 """ x = 0 **while** x < 4:  print(x)  x += 1 *# versión corta de x = x + 1* |
| --- |

Python 3 permite el manejo de excepciones mediante “try” y “catch” como se observa aquí:

| *# Maneja excepciones con un bloque try/except* **try**:  *# Usa raise para levantar un error*  **raise** IndexError("Este es un error de indice") **except** IndexError **as** e:  **pass** *# Pass no hace nada (“pasa”). Usualmente aquí harias alguna recuperacion.* |
| --- |

Aquí vemos un ejemplo de como crear elementos iterables y algunas propiedades. En el ejemplo, trabajaremos utilizando las “claves” (keys) de un diccionario y poder recorrerlos con un for.

| *# Python ofrece una abstracción fundamental llamada Iterable.* *# Un iterable es un objeto que puede ser tratado como una sequencia.* *# El objeto es retornado por la función 'range' es un iterable.*  dicc\_lleno = {"uno": 1, "dos": 2, "tres": 3} nuestro\_iterable = dicc\_lleno.keys() print(nuestro\_iterable) *# => dict\_keys(['uno', 'dos', 'tres']). Este es un objeto que implementa nuestra interfaz Iterable*  Podemos recorrerla. **for** i **in** nuestro\_iterable:  print(i) *# Imprime uno, dos, tres*  *# Aunque no podemos selecionar un elemento por su índice.* nuestro\_iterable[1] *# Genera un TypeError*  *# Un iterable es un objeto que sabe como crear un iterador.* nuestro\_iterator = iter(nuestro\_iterable)  *# Nuestro iterador es un objeto que puede recordar el estado mientras lo recorremos.* *# Obtenemos el siguiente objeto llamando la función \_\_next\_\_.* nuestro\_iterator.\_\_next\_\_() *# => "uno"*  *# Mantiene el estado mientras llamamos \_\_next\_\_.* nuestro\_iterator.\_\_next\_\_() *# => "dos"* nuestro\_iterator.\_\_next\_\_() *# => "tres"*  *# Después que el iterador ha retornado todos sus datos, da una excepción StopIterator.* nuestro\_iterator.\_\_next\_\_() *# Genera StopIteration*  *# Puedes obtener todos los elementos de un iterador llamando a list() en el.* list(dicc\_lleno.keys()) *# => Retorna ["uno", "dos", "tres"]* |
| --- |

## Funciones

| *####################################################* *## 4. Funciones* *####################################################* |
| --- |

Aquí algunos ejemplos de definición y llamada de funciones en Python 3.

| *# Usa 'def' para crear nuevas funciones* **def** **add**(x, y):  print("x es {} y y es {}".format(x, y))  **return** x + y *# Retorna valores con una la declaración return*  *# Llamando funciones con parámetros* add(5, 6) *# => imprime "x es 5 y y es 6" y retorna 11*  *# Otra forma de llamar funciones es con argumentos de palabras claves* add(y=6, x=5) *# Argumentos de palabra clave pueden ir en cualquier orden.*   *# Puedes definir funciones que tomen un número variable de argumentos* **def** **varargs**(\*args):  **return** args  varargs(1, 2, 3) *# => (1,2,3)*   *# Puedes definir funciones que toman un número variable de argumentos* *# de palabras claves* **def** **keyword\_args**(\*\*kwargs):  **return** kwargs  *# Llamémosla para ver que sucede* keyword\_args(pie="grande", lago="ness") *# => {"pie": "grande", "lago": "ness"}*   *# Puedes hacer ambas a la vez si quieres* **def** **todos\_los\_argumentos**(\*args, \*\*kwargs):  **print** args  **print** kwargs """ todos\_los\_argumentos(1, 2, a=3, b=4) imprime:  (1, 2)  {"a": 3, "b": 4} """  *# ¡Cuando llames funciones, puedes hacer lo opuesto a varargs/kwargs!* *# Usa \* para expandir tuplas y usa \*\* para expandir argumentos de palabras claves.* args = (1, 2, 3, 4) kwargs = {"a": 3, "b": 4} todos\_los\_argumentos(\*args) *# es equivalente a foo(1, 2, 3, 4)* todos\_los\_argumentos(\*\*kwargs) *# es equivalente a foo(a=3, b=4)* todos\_los\_argumentos(\*args, \*\*kwargs) *# es equivalente a foo(1, 2, 3, 4, a=3, b=4)* |
| --- |

Para facilitar algunas operaciones, Python permite tanto funciones definidas (de primera clase) como funciones anónimas. Estas funciones anónimas nos ayudan sobre todo a utilizar “programación funcional” con funciones como “map”, “filter” y “reduce”.

| *# Python tiene funciones de primera clase* **def** **crear\_suma**(x):  **def** **suma**(y):  **return** x + y  **return** suma  sumar\_10 = crear\_suma(10) sumar\_10(3) *# => 13*  *# También hay funciones anónimas* (**lambda** x: x > 2)(3) *# => True*  *# Hay funciones integradas de orden superior* map(sumar\_10, [1,2,3]) *# => [11, 12, 13]* filter(**lambda** x: x > 5, [3, 4, 5, 6, 7]) *# => [6, 7]*  *# Podemos usar listas por comprensión para mapeos y filtros agradables* [add\_10(i) **for** i **in** [1, 2, 3]] *# => [11, 12, 13]* [x **for** x **in** [3, 4, 5, 6, 7] **if** x > 5] *# => [6, 7]* *# también hay diccionarios* {k:k\*\*2 **for** k **in** range(3)} *# => {0: 0, 1: 1, 2: 4}* *# y conjuntos por comprensión* {c **for** c **in** "la cadena"} *# => {'d', 'l', 'a', 'n', ' ', 'c', 'e'}* |
| --- |

## Clases

| *####################################################* *## 5. Clases* *####################################################* |
| --- |

Las clases en Python, heredan inicialmente del objeto predefinido “object”. Aquí un ejemplo de definición de clase con atributos, constructor y métodos.

| *# Heredamos de object para obtener una clase.* **class** **Humano**(object):   *# Un atributo de clase es compartido por todas las instancias de esta clase*  especie = "H. sapiens"   *# Constructor basico*  **def** **\_\_init\_\_**(self, nombre):  *# Asigna el argumento al atributo nombre de la instancia*  self.nombre = nombre   *# Un metodo de instancia. Todos los metodos toman self como primer argumento*  **def** **decir**(self, msg):  **return** "%s: %s" % (self.nombre, msg)   *# Un metodo de clase es compartido a través de todas las instancias*  *# Son llamados con la clase como primer argumento*  **@classmethod**  **def** **get\_especie**(cls):  **return** cls.especie   *# Un metodo estatico es llamado sin la clase o instancia como referencia*  **@staticmethod**  **def** **roncar**():  **return** "\*roncar\*"   *# Instancia una clase* i = Humano(nombre="Ian") **print** i.decir("hi") *# imprime "Ian: hi"*  j = Humano("Joel") **print** j.decir("hello") *#imprime "Joel: hello"*  *# Llama nuestro método de clase* i.get\_especie() *# => "H. sapiens"*  *# Cambia los atributos compartidos* Humano.especie = "H. neanderthalensis" i.get\_especie() *# => "H. neanderthalensis"* j.get\_especie() *# => "H. neanderthalensis"*  *# Llama al método estático* Humano.roncar() *# => "\*roncar\*"* |
| --- |
|  |

## Módulos

| *####################################################* *## 6. Módulos* *####################################################* |
| --- |

Python permite importar módulos, tanto creados por nosotros, como existentes en el sistema. Una potente herramienta para descargar módulos más populares es “pip” <https://pypi.org/project/pip/>

| *# Puedes importar módulos* **import** math print(math.sqrt(16)) *# => 4.0*  *# Puedes obtener funciones específicas desde un módulo* **from** math **import** ceil, floor print(ceil(3.7)) *# => 4.0* print(floor(3.7))*# => 3.0*  *# Puedes importar todas las funciones de un módulo* *# Precaución: Esto no es recomendable* **from** math **import** \*  *# Puedes acortar los nombres de los módulos* **import** math **as** m math.sqrt(16) == m.sqrt(16) *# => True*  *# Los módulos de Python son sólo archivos ordinarios de Python.* *# Puedes escribir tus propios módulos e importarlos. El nombre del módulo* *# es el mismo del nombre del archivo.*  *# Puedes encontrar que funciones y atributos definen un módulo.* **import** math dir(math) |
| --- |

## Avanzado: generadores y decoradores

Ejemplo de creación de generadores:

| *# Los generadores te ayudan a hacer un código perezoso (lazy)* **def** **duplicar\_numeros**(iterable):  **for** i **in** iterable:  **yield** i + i  *# Un generador crea valores sobre la marcha.* *# En vez de generar y retornar todos los valores de una vez, crea uno en cada iteración.* *# Esto significa que valores más grandes que 15 no serán procesados en 'duplicar\_numeros'.* *# Fíjate que 'range' es un generador. Crear una lista 1-900000000 tomaría mucho tiempo en crearse.* \_rango = range(1, 900000000) *# Duplicará todos los números hasta que un resultado >= se encuentre.* **for** i **in** duplicar\_numeros(\_rango):  print(i)  **if** i >= 30:  **break** |
| --- |

Ejemplo de utilización de decoradores en Python.

| *# Decoradores* *# en este ejemplo 'pedir' envuelve a 'decir'* *# Pedir llamará a 'decir'. Si decir\_por\_favor es True entonces cambiará el mensaje a retornar* **from** functools **import** wraps   **def** **pedir**(\_decir):  **@wraps(\_decir)**  **def** **wrapper**(\*args, \*\*kwargs):  mensaje, decir\_por\_favor = \_decir(\*args, \*\*kwargs)  **if** decir\_por\_favor:  **return** "{} {}".format(mensaje, "¡Por favor! Soy pobre :(")  **return** mensaje   **return** wrapper  **@pedir** **def** **say**(decir\_por\_favor=False):  mensaje = "¿Puedes comprarme una cerveza?"  **return** mensaje, decir\_por\_favor   print(decir()) *# ¿Puedes comprarme una cerveza?* print(decir(decir\_por\_favor=**True**)) *# ¿Puedes comprarme una cerveza? ¡Por favor! Soy pobre :()* |
| --- |

# 

# Usar Python para hacer Scripts de Sistemas Operativos

A continuación presentamos algunos consejos útiles para trabajar con Python 3 a la hora de realizar scripts de sistemas operativos.

**Consejo 1)** Prácticamente, todas las distribuciones de Linux incluyen Python. Para saber donde está podéis ejecutar el comando:

| which python |
| --- |

Y os dará la ruta. En la gran mayoría de estas su ruta es “/usr/bin/python3”.

Para hacer un ShellScript en Python, debéis incluir en la primera línea que el intérprete a usar sea

Python, podéis hacerlo en

| *#!/usr/bin/python3* |
| --- |

Después, al igual que cualquier otro tipo de ShellScript, deberéis darle permisos de ejecución, por

ejemplo con el comando:

| chmod u+x ./nuestroScript.py |
| --- |

NOTA: la extensión .py, aunque recomendable por temas de comprensión, no es obligatoria.

**Consejo 2)** Ejecutar comandos Linux y obtener su salida por pantalla

| **import** commands status, output = commands.getstatusoutput("cat /etc/services") |
| --- |

Este comando devuelve el status de la ejecución en la variable “status” (generalmente 0 si él

comando se ha ejecutado correctamente y distintos de 0 si ha habido un error, indicando cuál) y la

salida que ser vería por pantalla (la salida estándar) se almacena en la variable output.

**Consejo 3)** Leer de teclado (similar a la función “read” en Bash ShellScript)

De manera simple se puede hacer con

| nb = input('Elige un número: ') |
| --- |

Y se guardará lo leído por teclado en la variable “nb”.

**Consejo 4)** Para el paso de parámetros desde consola en Python, hay que importar “sys” y usar “sys.argv”. Esta variable es un vector que contiene el nombre del ejecutable en la posición 0 y en las siguientes posiciones los parámetros en orden

| **import** sys **if** len(sys.argv) != 3:   **print** "Se requieren 2 parámetros"  **else**:  **print** sys.argv[0] *# devuelve el nombre del ejecutable.*  **print** sys.argv[1] *# devuelve el primer argumento*  **print** sys.argv[2] *# devuelve el segundo argumento* |
| --- |

**Consejo 5)** Simular la herramienta test (para comprobar si existen ficheros, directorios, etc.)

Por ejemplo, para saber si una ruta existe, sea fichero o directorio:

| **import** os.path file\_path="/home" **if** os.path.exists(file\_path):  **print** "La ruta existe" **else**:  **print** "La ruta no existe" |
| --- |

Por ejemplo, para saber si es un directorio (no fichero)

| **import** os.path file\_path="/home" **if** os.path.isdir(file\_path):  **print** "La ruta es un directorio" **else**:  **print** "La ruta no existe o no es un directorio" |
| --- |

Por ejemplo, comprobar si una ruta es fichero (no directorio) y además comprobar si tenemos

permisos de lectura:

| **import** os **import** os.path PATH='./file.txt' **if** os.path.isfile(PATH) **and** os.access(PATH, os.R\_OK):  **print** "File exists and is readable" **else**:  **print** "Either file is missing or is not readable" |
| --- |

Para usar con “os.access”:

* **os.R\_OK**: Para comprobar si se tiene permiso de lectura en la ruta.
* **os.W\_OK:** Para comprobar si se tiene permiso de escritura en la ruta.
* **os.X\_OK**: Para comprobar si se tiene permiso de ejecución en la ruta.

**Consejo 6)** Leer variables de entorno de Linux desde Python:

| **import** os **print** os.environ['HOME'] *# Esto nos imprime la variable de entorno de Linux "HOME"* |
| --- |

# Bibliografía

* Learn X in Y Minutes: <https://learnxinyminutes.com/docs/es-es/python-es/>
* Aprende Python con Alf <https://aprendeconalf.es/docencia/python/manual/>
* Python para todos: <http://do1.dr-chuck.com/pythonlearn/ES_es/pythonlearn.pdf>