

Python 1.

Instalar python.

Descargar python de : <https://www.python.org/downloads/>

Existen dos formas de ejecutar código Python:

- **Sesión interactiva:** podemos escribir líneas de código en el intérprete y obtener una respuesta. Para iniciar el intérprete en línea de comandos escribimos `python3`, para salir de él, escribir `exit()` o bien **Control +D**.
- **Archivo:** podemos escribir el código de un programa en un archivo de texto y ejecutarlo, extensión `.py`.

En windows para ejecutar un programa python (.py) dado que los archivos .py ya están asociados al intérprete de Python, solo basta con hacer doble clic sobre el archivo para ejecutar el programa.

En Linux para que el sistema operativo abra el archivo .py con el intérprete adecuado, es necesario añadir una nueva línea al principio del archivo.

Shebang:

Es la primera línea de un archivo python : `#!/usr/bin/python`

El par de caracteres `#!` indica al sistema operativo que dicho script se debe ejecutar utilizando el intérprete especificado, esto depende de la ruta donde está instalado nuestro intérprete. Otra opción es : `#!/usr/bin/env python`

Una vez añadido shebang ejecutamos el programa en línea de comandos : `./hola.py`

Es importante dar permisos de ejecución: `chmod +x hola.py`

Gestión de paquetes:

La instalación de Python ya ofrece muchos paquetes y módulos por defecto, es lo que se llama Librería estándar, pero podemos instalar muchos más.

Para gestionar los paquetes tenemos en nuestro sistema la herramienta **pip**, una utilidad incluida en la instalación de Python con la que podemos instalar, desinstalar y actualizar paquetes.

Ejemplo:

```
$ pip install pandas
```

IDE.

Podemos usar tanto un editor de texto básico o algunos más completos como:

pyDEV, SOE, Eric, Boa Constructor, Vim, Atom, notepad ++, sublimeText, Visual Sutdio, Eclipse +Pydev, Tonny, Jupyter Notebook, repl.it (servicio web), WSL (consola con entorno Linux para usar en windows)....

Un editor para principiantes muy completo es [Thonny](#).

Para trabajar con Python debemos tener instalado un intérprete, un editor y es interesante un depurador o debugger, Thonny engloba estas tres herramientas.

Tipos de datos.

- **Tipado dinámico:** La característica de tipado dinámico se refiere a que no es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo de valor al que se asigne, y el tipo de esta variable puede cambiar si se asigna un valor de otro tipo.
- **Fuertemente tipado:** no se permite tratar a una variable como si fuera un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente.

Nombre	Tipo	Ejemplos
Booleano	bool	<code>True, False</code>
Entero	int	<code>21, 34500, 34_500</code>
Flotante	float	<code>3.14, 1.5e3</code>
Complejo	complex	<code>2j, 3 + 5j</code>
Cadena	str	<code>'tfn', '''tenerife - islas canarias'''</code>
Tupla	tuple	<code>(1, 3, 5)</code>
Lista	list	<code>['Chrome', 'Firefox']</code>
Conjunto	set	<code>set([2, 4, 6])</code>
Diccionario	dict	<code>{'Chrome': 'v79', 'Firefox': 'v71'}</code>

Type: usamos type para saber el tipo de una determianda variable.

Ejemplo:

```
entero = 23
type(entero) devolvería int.
```

Variables.

En Python existen una serie de reglas para los nombres de variables:

1. **Sólo pueden contener los siguientes caracteres:**

- Letras minúsculas o mayúsculas
- Dígitos
- Guiones bajos (_).

2. Deben **empezar con una letra o un guión bajo**, nunca con un dígito.

3. No pueden ser una **palabra reservada** del lenguaje («keywords»).

4. Los nombres de variables son «case-sensitive».

Las **variables** son fundamentales ya que permiten definir **nombres** para los **valores** que tenemos en memoria y que vamos a usar en nuestro programa. No es necesario indicar el tipo de la variable cuando las declaramos.



nombre = valor

Operadores aritméticos.

- $a + b$ = Suma.
- $a - b$ = Resta a menos b.
- $a * b$ = Multiplicación.
- a / b = División.
- $a ** b$ = Exponente
- $a // b$ = División entera.
- $a \% b$ = Módulo.

Operadores lógicos o condicionales.

- **and** $x \text{ and } y$
- **or** $x \text{ or } y$
- **not** $\text{not } x$
- **==** $x == y$ son iguales ?
- **!=** $x != y$ son distintos ?
- **<** $a < b$ es a menor que b?
- **>** $a > b$ es a mayor que b ?
- **<=** menor o igual
- **>=** mayor o igual

Type: usamos type para saber el tipo de una determinada variable.

```
>>> is_raining = False
>>> type(is_raining)
bool

>>> sound_level = 35
>>> type(sound_level)
int

>>> temperature = 36.6
>>> type(temperature)
float
```

Igualmente existe la posibilidad de **comprobar el tipo** que tiene una variable mediante la función `isinstance()`:

```
>>> isinstance(is_raining, bool)
True
>>> isinstance(sound_level, int)
True
>>> isinstance(temperature, float)
True
```

Conversión implícita y explícita.

Cuando mezclamos enteros, booleanos y flotantes, Python realiza automáticamente una **conversión implícita** de los valores al tipo de «mayor rango». Veamos algunos ejemplos de esto:

```
>>> True + 25
26
>>> 7 * False
0
>>> True + False
1
>>> 21.8 + True
22.8
>>> 10 + 11.3
21.3
```

Podemos resumir la conversión implícita en la siguiente tabla:

Tipo 1	Tipo 2	Resultado
bool	int	int
bool	float	float
int	float	float

Aunque más adelante veremos el concepto de función, desde ahora podemos decir que existen una serie de funciones para realizar [conversiones explícitas](#) de un tipo a otro:

- **bool()** Convierte el tipo a booleano.
- **Int()** Convierte el tipo a entero.
- **Float()** Convierte el tipo a flotante.
- **Str()** convierte a string.

Veamos algunos ejemplos de estas funciones:

```

>>> bool(1)
True
>>> bool(0)
False
>>> int(True)
1
>>> int(False)
0
>>> float(1)
1.0

```

Importante:

usaremos mucho `str()` cuando queremos mostrar por pantalla una variable de un tipo que no sea string usando `print()`.

Ejemplo:

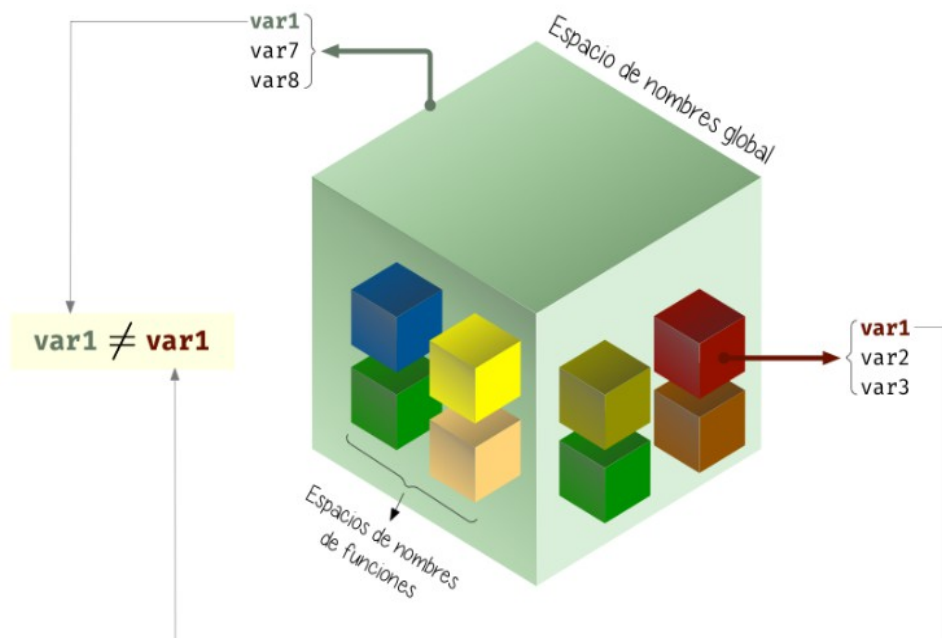
```

resultado = 29
Print("el resultado es: "+str(resultado))

```

Variables locales y globales.

Cada *función* define su propio espacio de nombres y es diferente del espacio de nombres global aplicable a todo nuestro programa.



Acceso a variables globales

Cuando una variable se define en el espacio de nombres global podemos hacer uso de ella con total transparencia dentro del ámbito de las funciones del programa:

```
>>> language = 'castellano'

>>> def catalonia():
...     print(f'{language=}')
...

>>> language
'castellano'

>>> catalonia()
language='castellano'
```

Creando variables locales

En el caso de que asignemos un valor a una variable global dentro de una función, no estaremos modificando ese valor. Por el contrario, estaremos creando una variable en el espacio de nombres local:

```
>>> language = 'castellano'

>>> def catalonia():
...     language = 'catalan'
...     print(f'{language=}')
...

>>> language
'castellano'

>>> catalonia()
language='catalan'

>>> language
'castellano'
```

Forzando modificación global

Python nos permite modificar una variable definida en un espacio de nombres global dentro de una función. Para ello debemos usar el modificador *global*:


```
>>> language = 'castellano'

>>> def catalonia():
...     global language
...     language = 'catalan'
...     print(f'{language=}')
...

>>> language
'castellano'

>>> catalonia()
language='catalan'

>>> language
'catalan'
```

Advertencia

El uso de `global` no se considera una buena práctica ya que puede inducir a confusión y tener efectos colaterales indeseados.

Contenido de los espacios de nombres

Python proporciona dos funciones para acceder al contenido de los espacios de nombres:

`locals()`

Devuelve un diccionario con los contenidos del **espacio de nombres local**.

`globals()`

Devuelve un diccionario con los contenidos del **espacio de nombres global**.

```

>>> language = 'castellano'

>>> def catalonia():
...     language = 'catalan'
...     print(f'{locals()}')
...

>>> language
'castellano'

>>> catalonia()
locals()={'language': 'catalan'}

>>> globals()
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 'loader': None.

```

Entrada por teclado: la función `input()`

La función `input()` permite obtener texto escrito por teclado. Al llegar a la función, el programa se detiene esperando que se escriba algo y se pulse la tecla **Intro**, como muestra el siguiente ejemplo:

```

print("¿Cómo se llama?")
nombre = input()
print(f"Me alegro de conocerle, {nombre}")

```

```

¿Cómo se llama?
Pepe
Me alegro de conocerle, Pepe

```

En el ejemplo anterior, el usuario escribe su respuesta en una línea distinta a la pregunta porque Python añade un salto de línea al final de cada `print()`.

Si se prefiere que el usuario escriba su respuesta a continuación de la pregunta, se podría utilizar el argumento opcional `end` en la función `print()`, que indica el carácter o caracteres a utilizar en vez del salto de línea. Para separar la respuesta de la pregunta se ha añadido un espacio al final de la pregunta.

```
print("¿Cómo se llama? ", end=" ")
nombre = input()
print(f"Me alegro de conocerle, {nombre}")
```

```
¿Cómo se llama? Pepe
Me alegro de conocerle, Pepe
```

Otra solución, más compacta, es aprovechar que a la función `input()` se le puede enviar un argumento que se escribe en la pantalla (sin añadir un salto de línea):

```
nombre = input("¿Cómo se llama? ")
print(f"Me alegro de conocerle, {nombre}")
```

```
¿Cómo se llama? Pepe
Me alegro de conocerle, Pepe
```

Conversión de tipos

De forma predeterminada, la función `input()` convierte la entrada en una cadena, aunque escribamos un número. Si intentamos hacer operaciones, se producirá un error.

Si se quiere que Python interprete la entrada como un número entero, se debe utilizar la función `int()` de la siguiente manera:

```
cantidad = int(input("Dígame una cantidad en pesetas: "))
print(f"{cantidad} pesetas son {round(cantidad / 166.386, 2)} euros")
```

```
Dígame una cantidad en pesetas: 500
500 pesetas son 3.01 euros
```

Salida por pantalla: la función `print()`

En los programas, para que python nos muestre texto o variables hay que utilizar la función `print()`.

La función `print()` permite mostrar texto en pantalla. El texto a mostrar se escribe como argumento de la función:

Las cadenas se pueden delimitar tanto por comillas dobles (") como por comillas simples (').

```
print("Hola")
```

```
Hola
```

La función `print()` admite varios argumentos seguidos. En el programa, los argumentos deben separarse por comas. Los argumentos se muestran en el mismo orden y en la misma línea, separados por espacios:

```
print("Hola", "Adiós")
```

```
Hola Adiós
```

Al final de cada `print()`, Python añade automáticamente un salto de línea:

```
print("Hola")  
print("Adiós")
```

```
Hola  
Adiós
```

Si se quiere que Python **no** añada un salto de línea al final de un `print()`, se debe añadir al final el argumento `end=""`:

```
print("Hola", end="")  
print("Adiós")
```

```
HolaAdios
```

En el ejemplo anterior, las dos cadenas se muestran pegadas. Si se quieren separar los argumentos en la salida, hay que incluir los espacios deseados (bien en la cadena, bien en el argumento `end`):

```
print("Hola.", end="")  
print("Adiós")
```

Hola. Adiós

```
print("Hola.", end="")  
print("Adiós")
```

Hola. Adiós

El valor del parámetro end puede ser una cadena f:

```
texto = " y "  
print("Hola", end=f"{texto}")  
print("Adiós")
```

Hola y Adiós

Como las comillas indican el principio y el final de una cadena, si se escriben comillas dentro de comillas se produce un error de sintaxis.

Para incluir comillas dentro de comillas, se puede escribir una contrabarra (\) antes de la comilla para que Python reconozca la comilla como carácter, no como delimitador de la cadena:

```
print("Un tipo le dice a otro: \"¿Cómo estás?\")  
print('Y el otro le contesta: \'¡Pues anda que tú!\'')
```

```
Un tipo le dice a otro: "¿Cómo estás?"  
Y el otro le contesta: '¡Pues anda que tú!'
```

O escribir comillas distintas a las utilizadas como delimitador de la cadena:

```
print("Un tipo le dice a otro: '¿Cómo estás?'")  
print('Y el otro le contesta: "¡Pues anda que tú!"')
```

```
Un tipo le dice a otro: '¿Cómo estás?'  
Y el otro le contesta: "¡Pues anda que tú!"
```

La función `print()` permite incluir variables o expresiones como argumento, lo que nos permite combinar texto y variables:

```
nombre = "Pepe"
edad = 25
print("Me llamo", nombre, "y tengo", edad, "años.")
```

Me llamo Pepe y tengo 25 años.

```
semanas = 4
print("En", semanas, "semanas hay", 7 * semanas, "días.")
```

En 4 semanas hay 28 días.

La función `print()` muestra los argumentos separados por espacios, lo que a veces no es conveniente. En el ejemplo siguiente el signo de exclamación se muestra separado de la palabra.

```
nombre = "Pepe"
print("¡Hola,", nombre, "!")
```

¡Hola, Pepe!

Antes de Python 3.6 estas situaciones se podían resolver de una forma algo engorrosa, pero a partir de Python 3.6 se pueden utilizar [las cadenas "f"](#).

```
nombre = "Pepe"
print(f"¡Hola, {nombre}!")
```

¡Hola, Pepe!

Strings.

Para escribir una cadena de texto en Python usamos comillas simples (') o bien comillas dobles (""), dentro de estas comillas se pueden añadir caracteres especiales escapandolos con \, como por ejemplo \n que indica el carácter de nueva línea o \t que indica tabulación.

Concatenar: usamos el simbolo + o *

Ejemplo:	a = "uno"	b="dos"
	c = a + b	c es "unodos"
	c = a * 3	C es "unounouno"

Cadenas raw: en ellas los caracteres escapados mediante la barra invertida (\) no se sustituyen por

sus contrapartidas, esto lo veremos en el capítulo de expresiones regulares.

Ejemplo:

<code>text1 = 'abc\ndef'</code>	<code>print(text1)</code>	muestra: abc def
<code>text2 = r'abc\ndef'</code>	<code>print(text2)</code>	muestra: abc\ndef

Varias líneas: podemos crear cadenas de varias líneas con comillas triples, de esta forma podemos escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea sin tener que recurrir al carácter `\n`.

Imprimir en pantalla. Print.

Usamos la función `print()`

Ejemplo: `print("hola mundo")`

Parámetros de `print`:

`print(msg1, msg2, sep='|')` definir un separador entre las cadenas.

`print(msg1, msg2, end='!!!')` definir un final de cadena.

Formatear String: **format**

```
>>> name = "Manny"
>>> number = len(name) * 3
>>> print("Hello {}, your lucky number is {}".format(name, number))
Hello Manny, your lucky number is 15
>>> print("Your lucky number is {number}, {name}.".format(name=name, number=len(name)*3))
Your lucky number is 15, Manny.
```

Formatting expressions

Expr	Meaning	Example
{:d}	integer value	'{:d}'.format(10.5) → '10'
{:.2f}	floating point with that many decimals	'{:2f}'.format(0.5) → '0.50'
{:2s}	string with that many characters	'{:2s}'.format('Python') → 'Py'
{:<6s}	string aligned to the left that many spaces	'{:<6s}'.format('Py') → 'Py '
{:>6s}	string aligned to the right that many spaces	'{:>6s}'.format('Py') → ' Py'
{:^6s}	string centered in that many spaces	'{:^6s}'.format('Py') → ' Py '

```
>>> price = 7.5
>>> with_tax = price * 1.09
>>> print(price, with_tax)
7.5 8.175
>>> print("Base price: ${:.2f}. With Tax: ${:.2f}".format(price, with_tax))
Base price: $7.50. With Tax: $8.18
>>> █
```

Formatting expression

Caracteres especiales

Los caracteres especiales empiezan por una contrabarra (\).

- Comilla doble: \"

```
>>> print("Las comillas dobles \" delimitan cadenas.")  
Las comillas dobles " delimitan cadenas.
```

- Comilla simple: \'

```
>>> print('Las comillas simples \' delimitan cadenas.')  
Las comillas simples ' delimitan cadenas.
```

- Salto de línea: \n

```
>>> print("Una línea\nOtra línea")  
Una línea  
Otra línea
```

- Tabulador: \t

```
>>> print("1\t2\t3")  
1      2      3
```



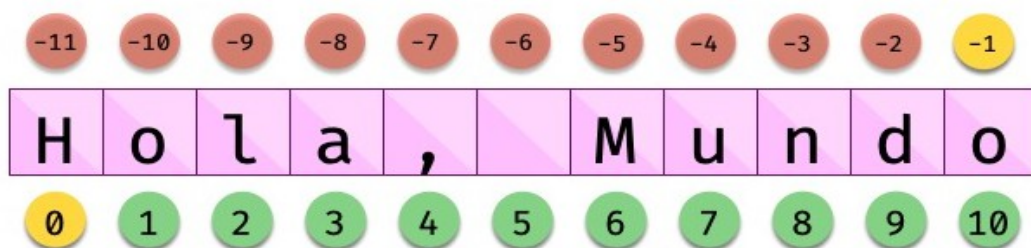
Por completar con todos los caracteres especiales

<https://www.mclibre.org/consultar/python/lecciones/python-cadenas.html#cadenas-f>

String indexing.

Los «strings» están **indexados** y cada carácter tiene su propia posición. Para obtener un único carácter dentro de una cadena de texto es necesario especificar su **índice** dentro de corchetes [...].

Téngase en cuenta que el indexado de una cadena de texto siempre empieza en **0** y termina en **una unidad menos de la longitud** de la cadena.




```
>>> sentence = 'Hola, Mundo'

>>> sentence[0]
'H'
>>> sentence[-1]
'o'
>>> sentence[4]
','
>>> sentence[-5]
'M'
```

Longitud de la cadena.

Usamos la función `len()`.

```
>>> proverb = 'Lo cortés no quita lo valiente'
>>> len(proverb)
27
```

Slice o subcadena.

Es posible extraer «trozos» («rebanadas») de una cadena de texto. Tenemos varias aproximaciones para ello:

- | | |
|-------------------------|--|
| [:] | Extrae la secuencia entera desde el comienzo hasta el final. |
| [start:] | Extrae desde start hasta el final de la cadena. |
| [:end] | Extrae desde el comienzo de la cadena hasta end menos 1. |
| [start:end] | Extrae desde start hasta end menos 1. |
| [start:end:step] | Extrae desde start hasta end menos 1 haciendo saltos de tamaño step. |

Ejemplos:


```
>>> proverb = 'Agua pasada no mueve molino'

>>> proverb[:]
'Agua pasada no mueve molino'

>>> proverb[12:]
'no mueve molino'

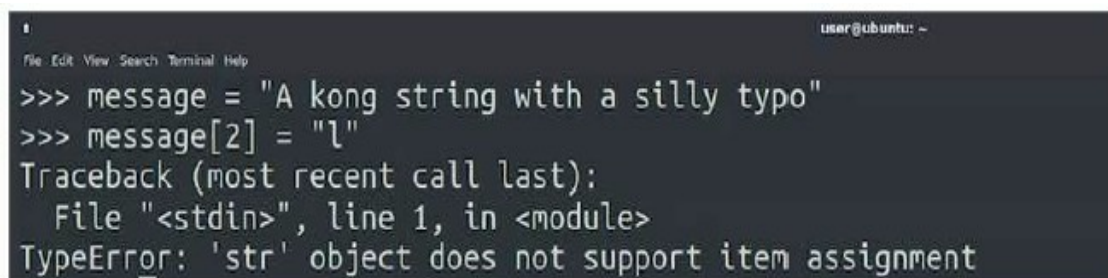
>>> proverb[:11]
'Agua pasada'

>>> proverb[5:11]
'pasada'

>>> proverb[5:11:2]
'psd'
```

Modificar cadena.

Las cadenas de texto son tipos de datos **inmutables**. Es por ello que no podemos modificar un carácter directamente.



```
user@ubuntu: ~
File Edit View Search Terminal Help
>>> message = "A kong string with a silly typo"
>>> message[2] = "l"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Lo haremos de la siguiente manera, usando una cadena auxiliar.

```
>>> new_message = message[0:2] + "l" + message[3:]
>>> print(new_message)
A long string with a silly typo
>>>
```

Reemplazar elementos.

Podemos usar la función `replace()` indicando la *subcadena a reemplazar*, la *subcadena de reemplazo* y *cuántas instancias* se deben reemplazar. Si no se especifica este último argumento, la sustitución se hará en todas las instancias encontradas:

```
>>> proverb = 'Quien mal anda mal acaba'

>>> proverb.replace('mal', 'bien')
'Quien bien anda bien acaba'

>>> proverb.replace('mal', 'bien', 1) # sólo 1 reemplazo
'Quien bien anda mal acaba'
```

Índice de un carácter.

Podemos obtener el índice de un determinado elemento de la cadena usando `index()`.

```
File Edit View Search Terminal Help
>>> pets = "Cats & Dogs"
>>> pets.index("&")
5
>>>
```

El método `index` da error si no encuentra la cadena buscada por eso podemos preguntar antes si la cadena está en nuestro string.

```
>>> "Dragons" in pets
False
```

Pertenencia de un elemento.

Para comprobar que una subcadena se encuentra en una cadena de texto utilizamos el operador `in`.

```
>>> proverb = 'Más vale malo conocido que bueno por conocer'

>>> 'malo' in proverb
True

>>> 'bueno' in proverb
True

>>> 'regular' in proverb
False
```

También podemos comprobar que una subcadena no está en la cadena.

```
>>> dna_sequence = 'ATGAAATTGAAATGGGA'

>>> not('C' in dna_sequence) # Primera aproximación
True

>>> 'C' not in dna_sequence # Forma pitónica
True
```

Dividir la cadena.

Usamos `split` para obtener una lista con todas las palabras de la cadena, podemos especificar el separador, si no se indica un separador, `split` toma por defecto los espacios en blanco, tabuladores y saltos de línea.

```
>>> proverb = 'No hay mal que por bien no venga'
>>> proverb.split()
['No', 'hay', 'mal', 'que', 'por', 'bien', 'no', 'venga']

>>> tools = 'Martillo,Sierra,Destornillador'
>>> tools.split(',')
['Martillo', 'Sierra', 'Destornillador']
```

Existe una forma algo más avanzada de dividir una cadena a través de **particionado**. Para ello podemos valernos de la función `partition()` que proporciona Python.

Esta función toma un argumento como separador, y divide la cadena de texto en 3 partes: lo que queda a la izquierda del separador, el separador en sí mismo y lo que queda a la derecha del separador:


```
>>> text = '3 + 4'

>>> text.partition('+')
('3 ', '+', ' 4')
```

Limpiar cadenas.

La función `strip()` se utiliza para eliminar caracteres del principio y del final de un «string».

Si no se especifican los caracteres a eliminar, `strip()` usa por defecto cualquier combinación de espacios en blanco, saltos de línea `\n` y tabuladores `\t`.

```
>>> serial_number = '\n\t  \n 48374983274832  \n\n\t  \t  \n'

>>> serial_number.strip()
'48374983274832'
```

También tenemos la versión para limpiar a izquierda y derecha.

Lstrip()

```
>>> serial_number.lstrip()
'48374983274832  \n\n\t  \t  \n'
```

rstrip()

```
>>> serial_number.rstrip()
'\n\t  \n 48374983274832'
```

Especificar los caracteres a borrar.

```
>>> serial_number.strip('\n')
'\t  \n 48374983274832  \n\n\t  \t  '
```

Comprobar si una cadena de texto [empieza o termina por alguna subcadena](#):

```
>>> lyrics.startswith('Quizás')
True

>>> lyrics.endswith('Final')
False
```

Encontrar la [primera ocurrencia](#) de alguna subcadena:

```
>>> lyrics.find('amor')
93

>>> lyrics.index('amor') # Same behaviour?
93
```

Tanto find() como index() devuelven el **índice** de la primera ocurrencia de la subcadena, pero se diferencian en su comportamiento cuando la subcadena buscada no existe:

```
>>> lyrics.find('universo')
-1

>>> lyrics.index('universo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Contabilizar el [número de veces que aparece](#) una subcadena:

```
>>> lyrics.count('mi')
2

>>> lyrics.count('tu')
3

>>> lyrics.count('él')
0
```

[Mayúsculas y minúsculas.](#)

```
>>> proverb = 'quien a buen árbol se arrima Buena Sombra le cobija'

>>> proverb
'quien a buen árbol se arrima Buena Sombra le cobija'

>>> proverb.capitalize()
'Quien a buen árbol se arrima buena sombra le cobija'

>>> proverb.title()
'Quien A Buen Árbol Se Arrima Buena Sombra Le Cobija'

>>> proverb.upper()
'QUIEN A BUEN ÁRBOL SE ARRIMA BUENA SOMBRA LE COBIJA'

>>> proverb.lower()
'quien a buen árbol se arrima buena sombra le cobija'

>>> proverb.swapcase()
'QUIEN A BUEN ÁRBOL SE ARRIMA BUENA sOMBRA LE COBIJA'
```

Identificando caracteres.

- `isalnum()` detectar si todos los caracteres son letras o números.
- `isnumeric()` si todos los caracteres son números.
- `isalpha()` si todos los caracteres son letras
- `isupper()` si está en mayúsculas.
- `islower()` si está en minúsculas.
- `istitle()` si está en formato de título.

Ejemplos.


```
>>> 'R2D2'.isalnum()
True
>>> 'C3-P0'.isalnum()
False

>>> '314'.isnumeric()
True
>>> '3.14'.isnumeric()
False

>>> 'abc'.isalpha()
True
>>> 'a-b-c'.isalpha()
False

>>> 'BIG'.isupper()
True
>>> 'small'.islower()
True
>>> 'First Heading'.istitle()
True
```

Desempaquetar una cadena.

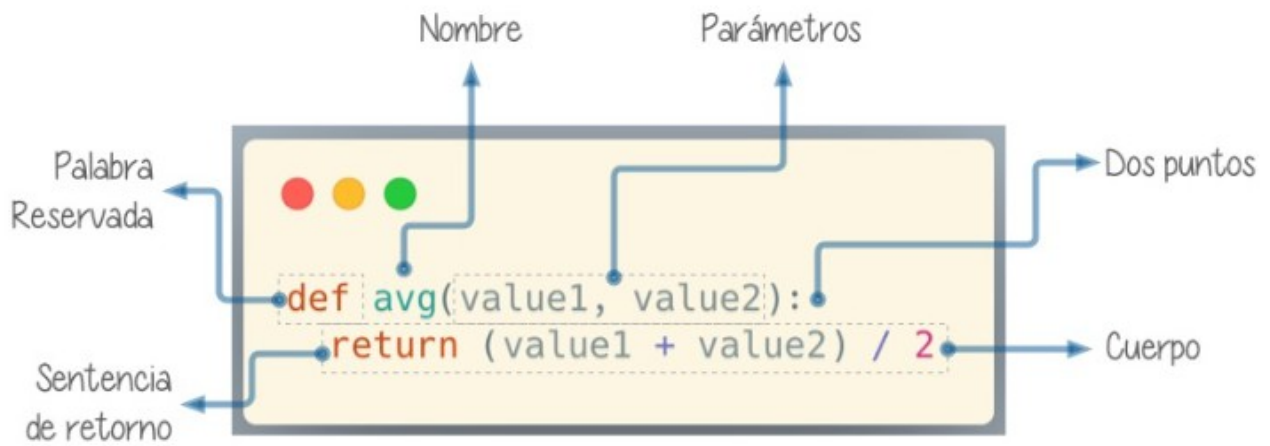

```
>>> oxygen = 'O2'
>>> first, last = oxygen
>>> first, last
('O', '2')

>>> text = 'Hello, World!'
>>> head, *body, tail = text
>>> head, body, tail
('H', ['e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd'], '!')
```

Funciones.

Definir una función.

Para definir una función utilizamos la palabra reservada **def** seguida del **nombre** de la función. A continuación aparecerán 0 o más **parámetros** separados por comas (entre paréntesis), finalizando la línea con **dos puntos :**. En la siguiente línea empezaría el **cuerpo** de la función que puede contener 1 o más **sentencias**, incluyendo (o no) una **sentencia de retorno** con el resultado mediante **return**.



Invocar una función.

Para invocar (o «llamar») a una función sólo tendremos que escribir su nombre seguido de paréntesis y el valor de los argumentos si los tiene.

Retornar un valor.

- Las funciones pueden devolver un valor, usando la palabra clave [return](#).
- En la sentencia return podemos incluir variables y expresiones, no únicamente literales.
- Si una función no incluye un return de forma explícita, devolverá None de forma implícita .

Ejemplos:


```
# función sin parámetros o retorno de valores
def diHola():
    print("Hello!")

diHola() # llamada a la función, 'Hello!' se muestra en la consola

# función con un parámetro
def holaConNombre(name):
    print("Hello " + name + "!")

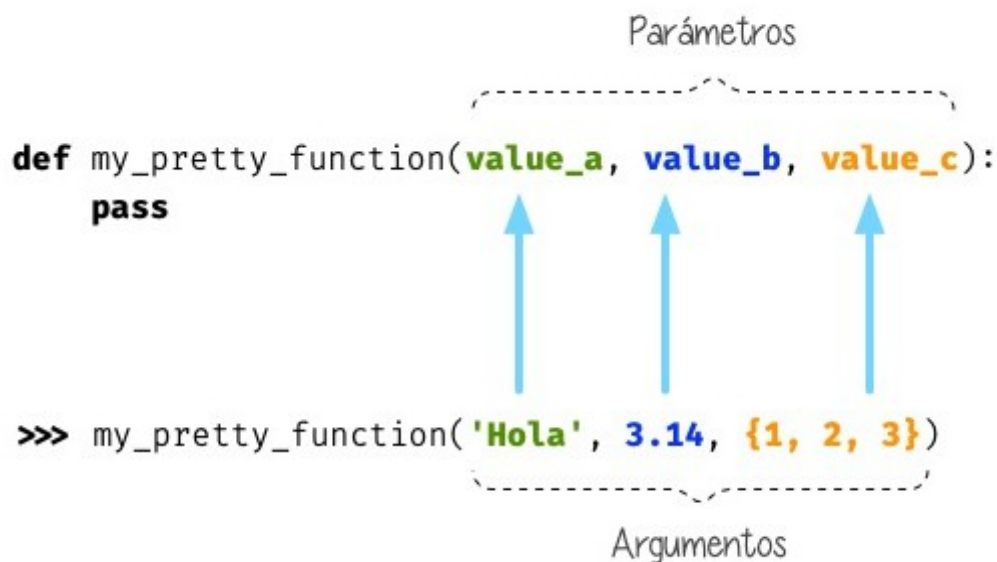
holaConNombre("Ada") # llamada a la función, 'Hello Ada!' se muestra en la consola

# función con múltiples parámetros con una sentencia de retorno
def multiplica(val1, val2):
    return val1 * val2

multiplica(3, 5) # muestra 15 en la consola
```

Parámetros y argumentos.

Cuando llamamos a una función con *argumentos*, los valores de estos argumentos se copian en los correspondientes *parámetros* dentro de la función:



Truco: La sentencia `pass` permite «no hacer nada». Es una especie de «*placeholder*».

Argumentos:

Posicionales:

Los **argumentos posicionales** son aquellos argumentos que se copian en sus correspondientes parámetros **en orden**. se necesita **recordar el orden** de los argumentos. Un error en la posición de los argumentos puede causar resultados indeseados.

Nominales:

En esta aproximación los argumentos no son copiados en un orden específico sino que **se asignan por nombre a cada parámetro**. Ello nos permite salvar el problema de conocer cuál es el orden de los parámetros en la definición de la función. Para utilizarlo, basta con realizar una asignación de cada argumento en la propia llamada a la función.

Ejemplos:

Dada la función:

```
>>> def build_cpu(vendor, num_cores, freq):  
...     return dict(  
...         vendor=vendor,  
...         num_cores=num_cores,  
...         freq=freq  
...     )
```

Llamada posicional:

```
>>> build_cpu('AMD', 8, 2.7)  
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Llamada nominal:

```
>>> build_cpu(vendor='AMD', num_cores=8, freq=2.7)  
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

```
>>> build_cpu(num_cores=8, freq=2.7, vendor='AMD')  
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Se puede ver claramente que el orden de los argumentos no influye en el resultado final.

Sin embargo, no es posible pasar un argumento de palabra clave antes que uno que no sea de palabra clave.

```
result = suma(3, b=2)
#result = 5
result2 = suma(b=2, 3)
#Lanzará SyntaxError
```

Parámetros por defecto.

Es posible especificar **valores por defecto** en los parámetros de una función. En el caso de que no se proporcione un valor al argumento en la llamada a la función, el parámetro correspondiente tomará el valor definido por defecto.

Ejemplo:

```
>>> def build_cpu(vendor, num_cores, freq=2.0):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
...     )
```

Llamada a la función sin especificar frecuencia de «cpu»:

```
>>> build_cpu('INTEL', 2)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 2.0}
```

Llamada a la función indicando una frecuencia concreta de «cpu»:

```
>>> build_cpu('INTEL', 2, 3.4)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 3.4}
```


Funciones con un número variable de argumentos.

Argumentos posicionales.

Si utilizamos el operador `*` delante del nombre de un parámetro posicional, estaremos indicando que los argumentos pasados a la función se empaqueten en una **tupla**.

```
>>> def _sum(*values):
...     result = 0
...     for value in values: # values es una tupla
...         result += arg
...     return result
...

>>> _sum(4, 3, 2, 1)
10
```

Existe la posibilidad de usar el asterisco `*` en la llamada a la función para **desempaquetar** los argumentos posicionales:

```
>>> def show_args(*args):
...     for arg in args:
...         print(f'{arg=}')
...

>>> my_args = (1, 2, 3, 4)

>>> show_args(my_args) # sin desempaquetado
arg=(1, 2, 3, 4)

>>> show_args(*my_args) # con desempaquetado
arg=1
arg=2
arg=3
arg=4
```

Nota

En muchas ocasiones se utiliza `args` como nombre del parámetro (es una convención).

Argumentos nominales.

Si utilizamos el operador `**` delante del nombre de un parámetro nominal, estaremos indicando que los argumentos pasados a la función se empaqueten en un **diccionario**.


```
>>> def best_student(**marks):
...     max_mark = -1
...     for student, mark in marks.items(): # marks es un diccionario
...         if mark > max_mark:
...             max_mark = mark
...             best_student = student
...     return best_student
...

>>> best_student(ana=8, antonio=6, inma=9, javier=7)
'inma'
```

Nota

En muchas ocasiones se utiliza `kwargs` como nombre del parámetro (es una convención).

Al igual que veíamos previamente, existe la posibilidad de usar doble asterisco `**` en la llamada a la función, para **desempaquetar** los argumentos nominales:

```
>>> def show_kwargs(**kwargs):
...     for item in kwargs.items():
...         print(f'{item=}')
...

>>> my_kwargs = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> show_kwargs(**my_kwargs)
item=('a', 1)
item=('b', 2)
item=('c', 3)
item=('d', 4)
```

Paso por valor y por referencia.

En el paso [por referencia](#) se pasa la dirección de memoria en la que se encuentra el contenido de la variable, y no el contenido en si.

En el paso [por valor](#), lo que se pasa como argumento es el valor que contiene la variable.

La diferencia entre ambos estriba en que el paso por valor los cambios que se hagan sobre el parámetro no se ven fuera de la función, si quisieramos modificar el valor de uno de los argumentos y que estos cambios se reflejaran fuera de la función tendríamos que pasar el parámetro por referencia.

En Python todo son objetos, cuando realizamos modificaciones a los argumentos de una función es importante tener en cuenta si son **mutables** (listas, diccionarios, conjuntos, ...) o **inmutables** (tuplas, enteros, flotantes, cadenas de texto, ...) ya que podríamos obtener efectos colaterales no deseados .

En resumen, los valores mutables se comportan como paso por referencia, y los inmutables como paso por valor.

Documentación o Docstring.

La primera línea del cuerpo de una función se conoce como docstring (cadena de documentación) y sirven, a modo de documentación de la función. Se suele usar comillas triples.

```
>>> def closest_int(value):
...     '''Returns the closest integer to the given value.
...     The operation is:
...         1. Compute distance to floor.
...         2. If distance less than a half, return floor.
...         Otherwise, return ceil.
...     '''
...     floor = int(value)
...     if value - floor < 0.5:
```

Para ver el docstring de una función, basta con utilizar **help**:

```
>>> help(closest_int)

Help on function closest_int in module __main__:

closest_int(value)
    Returns the closest integer to the given value.
    The operation is:
        1. Compute distance to floor.
        2. If distance less than a half, return floor.
        Otherwise, return ceil.
```

También es posible ver la información usando el símbolo de la interrogación:


```
>>> closest_int?
Signature: closest_int(value)
Docstring:
Returns the closest integer to the given value.
The operation is:
    1. Compute distance to floor.
    2. If distance less than a half, return floor.
    Otherwise, return ceil.
File:      ~/aprendepython/<ipython-input-75-5dc166360da1>
Type:      function
```

Explicación de parámetros.

Como ya se ha visto, es posible documentar una función utilizando un docstring. Pero la redacción y el formato de esta cadena de texto puede ser muy variada. Existen distintas formas de documentar una función (u otros objetos):

- [Sphinx docstrings](#): Formato nativo de documentación [Sphinx](#).
- [Google docstrings](#): Formato de documentación recomendado por Google.
- [NumPy-SciPy docstrings](#): Combinación de formatos reStructured y Google (usados por el proyecto [NumPy](#)).
- [Epytext](#): Una adaptación a Python de Epydoc(Java).

Aunque cada uno tienes sus particularidades, todos comparten una misma estructura:

- Una primera línea de **descripción de la función**.
- A continuación especificamos las características de los **parámetros** (incluyendo sus tipos).
- Por último, indicamos si la función **retorna un valor** y sus características.

Ejemplo 1:


```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> def to_seconds(hours, minutes, seconds):  
...     """Returns the amount of seconds in the given hours, minutes, and seconds."""  
...     return hours*3600+minutes*60+seconds  
...  
>>> help(to_seconds)
```

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
  
Help on function to_seconds in module __main__:  
  
to_seconds(hours, minutes, seconds)  
    Returns the amount of seconds in the given hours, minutes, and seconds.  
(END)
```

Ejemplo 2:

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> class Piglet:  
...     """Represents a piglet that can say their name."""  
...     years = 0  
...     name = ""  
...     def speak(self):  
...         """Outputs a message including the name of the piglet."""  
...         print("Oink! I'm {}! Oink!".format(self.name))  
...     def pig_years(self):  
...         """Converts the current age to equivalent pig years."""  
...         return self.years * 18  
...  
>>>
```

Ejemplo 3:

Remember our Person class from the last video? Let's add a docstring to the greeting method. How about, "Outputs a message with the name of the person".

```
1 class Person:
2
3     def __init__(self, name):
4         """ constructor de la clase """
5         self.name = name
6     def greeting(self):
7         """ Outputs a message with the name of the person"""
8         print("Hello! My name is {name}.".format(name=self.name))
9
10 help(Person)
```

Ejecutar

Restablecer


```

Here is your output:
Help on class Person in module submission:

class Person(builtins.object)
| Methods defined here:
|
|   __init__(self, name)
|       constructor de la clase
|
|   greeting(self)
|       Outputs a message with the name of the person
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
|
|
Excellent! You've mastered the art of providing info using
docstrings!

```

Ejemplo: Clase y su documentación usando la función Help, para ver la información básica de los métodos definidos en la clase.

```

4   ...         self.flavor = flavor
5   ...     def __str__(self):
6   ...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
7   ...
8   >>> help(Apple)
9   Help on class Apple in module __main__:
10
11  class Apple(builtins.object)
12  | Methods defined here:
13  |
14  |   __init__(self, color, flavor)
15  |       Initialize self. See help(type(self)) for accurate signature.
16  |
17  |   __str__(self)
18  |       Return str(self).
19  |
20  |   -----
21  |   Data descriptors defined here:
22  |
23  |   __dict__
24  |       dictionary for instance variables (if defined)
25  |
26  |   __weakref__
27  |       list of weak references to the object (if defined)

```

Condicionales.

if

```
if condición_1:
    bloque 1
elif condición_2:
    bloque 2
else:
    bloque 3
```

Valor nulo.

None es un valor especial de Python que almacena el **valor nulo** .

Para distinguir None de los valores propiamente booleanos, se recomienda el uso del operador is.

```
>>> value = None

>>> if value is None:
...     print('Value is clearly None')
... else:
...     # value podría contener True, False (u otro)
...     print('Value has some useful value')
...
Value is clearly void
```

```
>>> value = 99

>>> if value is not None:
...     print(f'{value=}')
...
value=99
```

Match-case.

Una de las novedades más esperadas (y quizás controvertidas) de Python 3.10 fue el llamado [Structural Pattern Matching](#) que introdujo en el lenguaje una nueva sentencia condicional. Ésta se podría asemejar a la sentencia «switch» que ya existe en otros lenguajes de programación.

¿Qué ocurre si el valor que comparamos no existe entre las opciones disponibles? Pues en principio, nada, ya que este caso no está cubierto. Si lo queremos controlar, hay que añadir una nueva regla utilizando el subguión `_` como patrón:

```
>>> color = '#AF549B'

>>> match color:
...     case '#FF0000':
...         print('●')
...     case '#00FF00':
...         print('●')
...     case '#0000FF':
...         print('●')
...     case _:
...         print('Unknown color!')
...
Unknown color!
```

```
>>> point = ('2', '5')

>>> match point:
...     case (int(), int()):
...         print(f'{point} is in plane')
...     case (int(), int(), int()):
...         print(f'{point} is in space')
...     case _:
...         print('Unknown!')
...
Unknown!
```

```
>>> point = ('8', 3, 5) # Nótese el 8 como "string"

>>> match point:
...     case (int(x), int(y)):
...         dist_to_origin = (x ** 2 + y ** 2) ** (1 / 2)
...     case (int(x), int(y), int(z)):
...         dist_to_origin = (x ** 2 + y ** 2 + z ** 2) ** (1 / 2)
...     case _:
...         print('Unknown!')
...
Unknown!
```

Operador morsa.

A partir de Python 3.8 se incorpora el operador morsa que permite unificar **sentencias de asignación dentro de expresiones**. Su nombre proviene de la forma que adquiere `:=`

```
>>> radius = 4.25
... if (perimeter := 2 * 3.14 * radius) < 100:
...     print('Increase radius to reach minimum perimeter')
...     print('Actual perimeter: ', perimeter)
...
Increase radius to reach minimum perimeter
Actual perimeter:  26.69
```

Bucles.

while

```
while condicion:
    cuerpo del bucle
```

Romper un bucle while

Python ofrece la posibilidad de romper o finalizar un bucle antes de que se cumpla la condición de parada. Usamos la instrucción `break`.

Python nos ofrece la posibilidad de **detectar si el bucle ha acabado de forma ordinaria**, esto es, ha finalizado por no cumplirse la condición establecida. Para ello podemos hacer uso de la sentencia `else` como parte del propio bucle. Si el bucle `while` finaliza normalmente (sin llamada a `break`) el flujo de control pasa a la sentencia opcional `else`.

```
>>> want_exit = 'N'
>>> num_questions = 0

>>> while want_exit == 'N':
...     print('Hola qué tal')
...     want_exit = input('¿Quiere salir? [S/N] ')
...     num_questions += 1
...     if num_questions == 4:
...         print('Máximo número de preguntas alcanzado')
...         break
...     else:
...         print('Usted ha decidido salir')
... print('Ciao')
Hola qué tal
¿Quiere salir? [S/N] S
Usted ha decidido salir
Ciao
```

[Continuar un bucle.](#)

Hay situaciones en las que, en vez de romper un bucle, nos interesa **saltar adelante hacia la siguiente repetición**. Para ello Python nos ofrece la sentencia `continue` que hace precisamente eso, descartar el resto del código del bucle y saltar a la siguiente iteración.


```

>>> want_exit = 'N'
>>> valid_options = 0

>>> while want_exit == 'N':
...     print('Hola qué tal')
...     want_exit = input('¿Quiere salir? [S/N] ')
...     if want_exit not in 'SN':
...         want_exit = 'N'
...         continue
...     valid_options += 1
...     print(f'{valid_options} respuestas válidas')
...     print('Ciao!')
Hola qué tal
¿Quiere salir? [S/N] N
Hola qué tal
¿Quiere salir? [S/N] X
Hola qué tal
¿Quiere salir? [S/N] Z
Hola qué tal
¿Quiere salir? [S/N] S
2 respuestas válidas
Ciao!

```

For.


```
for variable in elemento iterable (lista, cadena, range, etc.):  
    cuerpo del bucle
```

Python permite recorrer aquellos tipos de datos que sean **iterables**, es decir, que admitan *iterar* sobre ellos. Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (*recorridas*) son: cadenas de texto, listas, diccionarios, ficheros, etc. La sentencia for nos permite realizar esta acción.

Ejemplos.

```
>>> word = 'Python'  
  
>>> for letter in word:  
...     print(letter)  
...  
P  
y  
t  
h  
o  
n
```

```
import random  
  
print("Comienzo")  
cuenta_cincos = 0  
for i in range(3):  
    dado = random.randrange(1, 7)  
    print(f"Tirada {i + 1}: {dado}")  
    if dado == 5:  
        cuenta_cincos += 1  
print(f"En total ha(n) salido {cuenta_cincos} cinco(s).")  
print("Final")
```

[Romper un bucle for.](#)

```
>>> word = 'Python'

>>> for letter in word:
...     if letter == 't':
...         break
...     print(letter)
...
P
y
```

Truco

Tanto la [comprobación de rotura de un bucle](#) como la [continuación a la siguiente iteración](#) se llevan a cabo del mismo modo que hemos visto con los bucles de tipo `while`.

Secuencias de números. Range.

Es muy habitual hacer uso de secuencias de números en bucles. Python no tiene una instrucción específica para ello. Lo que sí aporta es una función `range()` que devuelve un flujo de números en el rango especificado. Una de las grandes ventajas es que la «lista» generada no se construye explícitamente, sino que cada valor se genera bajo demanda. Esta técnica mejora el consumo de recursos, especialmente en términos de memoria.

En Python 3, `range` es un tipo de datos. El tipo `range` es una **lista inmutable de números enteros en sucesión aritmética**.

- Inmutable significa que, a diferencia de las listas, los `range` no se pueden modificar.
- Una sucesión aritmética es una sucesión en la que la diferencia entre dos términos consecutivos es siempre la misma.

La técnica para la generación de secuencias de números es muy similar a la utilizada en los Slices de cadenas de texto. En este caso disponemos de la función `range(start, stop, step)`:

- **start**: Es opcional y tiene valor por defecto **0**.
- **stop**: es obligatorio (siempre se llega a 1 menos que este valor).
- **step**: es opcional y tiene valor por defecto **1**.

`range()` devuelve un objeto iterable, así que iremos obteniendo los valores paso a paso con una sentencia `for ... in`.

En resumen, los tres **argumentos del tipo `range(m, n, p)`** son:

- m: el valor inicial
- n: el valor final (que no se alcanza nunca)
- p: el paso (la cantidad que se avanza cada vez).

Si se escriben sólo dos argumentos, Python le asigna p el valor 1. Es decir `range(m, n)` es lo mismo que `range(m, n, 1)`

Si se escribe sólo un argumento, Python, le asigna a m el valor 0 y a p el valor 1. Es decir `range(n)` es lo mismo que `range(0, n, 1)`

El tipo `range()` sólo admite argumentos enteros. Si se utilizan argumentos decimales, se produce un error

Un `range` se crea llamando al tipo de datos con uno, dos o tres argumentos numéricos, como si fuera una función.

El tipo `range()` con un único argumento se escribe `range(n)` y crea una lista inmutable de n números enteros consecutivos que empieza en 0 y acaba en n - 1.

Para ver los valores del `range()`, es necesario convertirlo a lista mediante la función `list()`.

```
>>> x = range(10)
>>> x
range(0, 10)
>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(7)
range(0, 7)
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
```

Si *n* no es positivo, se crea un `range` vacío.

```
>>> list(range(-2))
[]
>>> list(range(0))
[]
```

El tipo `range` con dos argumentos se escribe `range(m, n)` y crea una lista inmutable de enteros consecutivos que empieza en m y acaba en $n - 1$.

```
>>> list(range(5, 1))
[]
>>> list(range(3, 3))
[]
```

Si n es menor o igual que m , se crea un `range` vacío.

```
>>> list(range(5, 1))
[]
>>> list(range(3, 3))
[]
```

El tipo `range` con tres argumentos se escribe `range(m, n, p)` y crea una lista inmutable de enteros que empieza en m y acaba justo antes de superar o igualar a n , aumentando los valores de p en p . Si p es negativo, los valores van disminuyendo de p en p .

```
>>> list(range(5, 21, 3))
[5, 8, 11, 14, 17, 20]
>>> list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

El valor de p no puede ser cero:

```
>>> range(4,18,0)
Traceback (most recent call last):/span>
  File "<pyshell#0>", line 1, in <module>
    range(4,18,0)
ValueError: range() arg 3 must not be zero
```

Si p es positivo y n menor o igual que m , o si p es negativo y n mayor o igual que m , se crea un

`range` vacío.

```
>>> list(range(25, 20, 2))  
[]  
>>> list(range(20, 25, -2))  
[]
```

En los `range(m, n, p)`, se pueden escribir p `range` distintos que generan el mismo resultado. Por ejemplo:


```
>>> list(range(10, 20, 3))
[10, 13, 16, 19]
>>> list(range(10, 21, 3))
[10, 13, 16, 19]
>>> list(range(10, 22, 3))
[10, 13, 16, 19]
```

Concatenar `range()`

No se pueden concatenar tipos `range()`, ya que el resultado de la concatenación puede no ser un tipo `range()`.

Pero sí se pueden concatenar tipos `range()` previamente convertidos en listas. El resultado es lógicamente una lista, que no se puede convertir a tipo `range()`.

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

La función `len()`

La función `len()` devuelve la longitud de una cadena de caracteres o el número de elementos de una lista. El argumento de la función `len()` es la lista o cadena que queremos "medir".

```
>>> len("mensaje secreto")
15
>>> len(["a", "b", "c"])
3
>>> len(range(1, 100, 7))
15
```

El valor devuelto por la función `len()` se puede usar como parámetro de `range()`.

```
>>> list(range(len("mensaje secreto")))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range(len(["a", "b", "c"])))
[0, 1, 2]
>>> list(range(len(range(1, 100, 7))))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Estructuras de datos.

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

Listas.

Las listas permiten almacenar objetos mediante un **orden definido** y con posibilidad de

duplicados. Las listas son estructuras de datos **mutables**, lo que significa que podemos añadir, eliminar o modificar sus elementos.

Crear listas.

En Python debemos escribir estos elementos separados por *comas* y dentro de *corchetes*.

```
>>> empty_list = []

>>> languages = ['Python', 'Ruby', 'Javascript']

>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]

>>> data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718, (28.2933947, -16.5226597)]
```

Nota

Una lista puede contener tipos de **datos heterogéneos**, lo que la hace una estructura de datos muy versátil.

Constructor.

También podemos crear una lista usando su constructor. `List()`.

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
```

Imprimir en pantalla.

Simplemente usamos `print(lista)`.

Conversión.

Para convertir otros tipos de datos en una lista podemos usar la función `list()`:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> # conversión desde una cadena de texto
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
```

Acceso un elemento.

Igual que en el caso de las cadenas de texto, podemos obtener un elemento de una lista a través del **índice** (lugar) que ocupa.

El índice comienza por cero.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[1]
'Huevos'

>>> shopping[2]
'Aceite'

>>> shopping[-1] # acceso con índice negativo
'Aceite'
```

También podemos **cambiar el valor** de un determinado elemento.


```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[0] = 'Jugo'

>>> shopping
['Jugo', 'Huevos', 'Aceite']
```

Dividir una lista.

Funciona igual que con las cadenas de texto.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[0:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[2:4]
['Aceite', 'Sal']
```

Importante

Ninguna de las operaciones anteriores modifican la lista original, simplemente devuelven una lista nueva.

Invertir una lista.

Conservando la lista original. [Reversed\(\)](#).

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> list(reversed(shopping))
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

Modificando la lista original. [Reverse\(\)](#).

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.reverse()

>>> shopping
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

[Añadir elementos.](#)

[Añadir al final de la lista. Append\(\)](#).

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.append('Atún')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Atún']
```

[Añadir en cualquier posición. Insert\(\)](#).

Inserta un elemento en la posición indicada sin machacar el existente, solo desplaza los elementos, si se añade en un índice superior simplemente se añade al final.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(1, 'Jamón')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Aceite']
```

Extend List

To append elements from another list to the current list, use the `extend()` method.

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

El resultado es la primera lista + la segunda lista, igual que concatenar.

Podemos usar el método `extend()` podemos usarlo con cualquier objeto iterable (tuples, sets, dictionaries etc.).

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

Repetir elementos.

Al igual que con las cadenas de texto, el operador `*` nos permite repetir los elementos de una lista.


```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping * 3
['Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite']
```

Combinar listas. Concatenar.

Conservando la lista original. Mediante el operador + =

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping + fruitshop
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

Modificando la lista original. Función extend().

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.extend(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

Otra forma es añadiendo todos los items de la lista 2 a la lista 1, uno por uno.

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
    list1.append(x)

print(list1)
```

Borrar elementos.

Por su índice. [Sentencia Del](#).

Borra el elemento que se indica mediante el índice.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> del shopping[3]

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

Con del también podemos borrar la lista por completo.

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Por su valor. [Función remove\(\)](#).

Elimina la primera ocurrencia del valor indicado en la lista.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.remove('Sal')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

Advertencia

Si existen valores duplicados, la función `remove()` sólo borrará la primera ocurrencia.

Por su índice y con extracción del elemento. [Pop\(\)](#).

Python nos ofrece la función `pop()` que además de borrar, nos «recupera» el elemento.

Eliminar elementos de la lista pero indicando un índice y nos devuelve el elemento eliminado, si no se indica el índice elimina el último elemento.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.pop()
'Limón'

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal']

>>> shopping.pop(2)
'Aceite'

>>> shopping
['Agua', 'Huevos', 'Sal']
```

Nota

Si usamos la función `pop()` sin pasarle ningún argumento, por defecto usará el índice `-1`, es decir, el último elemento de la lista. Pero también podemos indicarle el índice del elemento a extraer.

Por su rango. Mediante troceado de listas.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4] = []

>>> shopping
['Agua', 'Limón']
```

Borrado completo de la lista.

Función `clear`. elimina todos los elementos, la lista existe, pero está vacía.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.clear() # Borrado in-situ

>>> shopping
[]
```

Reiniciando a la [lista vacía](#).

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping = [] # Nueva zona de memoria

>>> shopping
[]
```

Encontrar un elemento. `Index()`.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.index('Huevos')
1
```

- Si el elemento no está en la lista no devolverá un error.
- Si buscamos un valor que existe más de una vez en la lista, solo obtendremos el índice de la primera ocurrencia.

Pertenencia de un elemento. `IN`.


```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> 'Aceite' in shopping
True

>>> 'Pollo' in shopping
False
```

Número de ocurrencias. Count().

```
>>> sheldon_greeting = ['Penny', 'Penny', 'Penny']

>>> sheldon_greeting.count('Howard')
0

>>> sheldon_greeting.count('Penny')
3
```

Longitud de una lista. Len().

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> len(shopping)
5
```

Convertir lista a cadena de texto.

Dada una lista, podemos convertirla a una cadena de texto, uniendo todos sus elementos mediante algún **separador**. Para ello hacemos uso de la función `join()` con la siguiente estructura:

`'=' . join(mylist)`
Separador Lista

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> ','.join(shopping)
'Agua,Huevos,Aceite,Sal,Limón'

>>> ' '.join(shopping)
'Agua Huevos Aceite Sal Limón'

>>> '|'.join(shopping)
'Agua|Huevos|Aceite|Sal|Limón'
```

Hay que tener en cuenta que `join()` sólo funciona si *todos sus elementos son cadenas de texto*.
[Ordenar una lista.](#)

Conservando la lista original. [Sorted\(\)](#).

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> sorted(shopping)
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

Modificando la lista. [Sort\(\)](#).

Método que ordenará la lista de manera alfanumerica, ascendente por defecto.

Es case sensitive, los resultados en mayúsculas van ordenados antes que en minúsculas.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.sort()

>>> shopping
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

Si queremos hacerlo de manera case-insensitive usamos `str.lower`.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

En orden descendente

```
thislist.sort(reverse = True)
```

Reverse orden por orden alfabético.

```
thislist.reverse()
```

Ambos métodos admiten un *parámetro* «booleano» `reverse` para indicar si queremos que la ordenación se haga en **sentido inverso** .

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> sorted(shopping, reverse=True)
['Sal', 'Limón', 'Huevos', 'Agua', 'Aceite']
```

Iteración sobre listas.

Usando la sentencia `for`.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> for product in shopping:
...     print(product)
...
Agua
Huevos
Aceite
Sal
Limón
```

Iterar usando enumeración.

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos **saber su índice** dentro de la misma. Para ello Python nos ofrece la función `enumerate()`.

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> for i, product in enumerate(shopping):
...     print(i, product)
...
0 Agua
1 Huevos
2 Aceite
3 Sal
4 Limón
```

Truco

Es posible utilizar el parámetro `start` con `enumerate()` para indicar el índice en el que queremos comenzar. Por defecto es 0.

Iterar sobre múltiples listas. Función `zip()`.


```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']

>>> for product, detail in zip(shopping, details):
...     print(product, detail)
...
Agua mineral natural
Aceite de oliva virgen
Arroz basmati
```

Nota

En el caso de que las listas no tengan la misma longitud, la función `zip()` realiza la combinación hasta que se agota la lista más corta.

- Usando índices:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
    print(thislist[i])
```

- usando bucle while:

```
thislist = ["apple", "banana", "cherry"]
i=0
while i < len(thislist):
    print(thislist[i])
    i = i+1
```

- usando List Comprehension:

```
thislist = ["apple", "banana", "cherry"]
print(x for x in thislist)
```

Cuidado con las copias.

Las listas son estructuras de datos mutables y esta característica nos obliga a tener cuidado cuando realizamos copias de listas, ya que la modificación de una de ellas puede afectar a la otra.


```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[15, 3, 7, 1]
```

Dado que las variables apuntan a la misma zona de memoria, al modificar una de ellas, el cambio también se ve reflejado en la otra.

Para solucionar esto podemos usar la función `copy()`:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list.copy()

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[4, 3, 7, 1]
```

Existe otra aproximación a este problema, y es utilizar un troceado completo de la lista, lo que nos devuelve una «copia desvinculada» de manera implícita.

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list[:]

>>> id(original_list) != id(copy_list)
True
```

Truco

En el caso de que estemos trabajando con listas que contienen elementos mutables, debemos hacer uso de la función `deepcopy()` dentro del módulo `copy` de la librería estándar.

Otra manera de hacer una copia es usar el constructor de la lista.

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Veracidad múltiple.

Si bien podemos usar sentencias condicionales para comprobar la veracidad de determinadas expresiones, Python nos ofrece dos funciones «built-in» con las que podemos evaluar si se cumplen todas las condiciones `all()` o si se cumple alguna condición `any()`. Estas funciones trabajan sobre iterables, y el caso más evidente es una lista.

Versión clásica.

```
>>> word = 'python'

>>> if len(word) > 4 and word.startswith('p') and word.count('y') >= 1:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Versión con all.

```
>>> word = 'python'

>>> enough_length = len(word) > 4           # True
>>> right_beginning = word.startswith('p')   # True
>>> min_ys = word.count('y') >= 1           # True

>>> is_cool_word = all([enough_length, right_beginning, min_ys])

>>> if is_cool_word:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Versión con any.


```

>>> word = 'yeah'

>>> enough_length = len(word) > 4           # False
>>> right_beginning = word.startswith('p')   # False
>>> min_ys = word.count('y') >= 1           # True

>>> is_fine_word = any([enough_length, right_beginning, min_ys])

>>> if is_fine_word:
...     print('Fine word!')
... else:
...     print('No thanks')
...
Fine word!

```

Desempaquetar una lista.

```

>>> writer1, writer2, writer3 = ['Virginia Woolf', 'Jane Austen', 'Mary Shelley']
>>> writer1, writer2, writer3
('Virginia Woolf', 'Jane Austen', 'Mary Shelley')

>>> text = 'Hello, World!'
>>> word1, word2 = text.split()
>>> word1, word2
('Hello,', 'World!')

```

List Methods

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

Tuplas.

Es similar a una lista, pero con la diferencia que no admite cambios, por lo tanto es **immutable**.

En Python, una **tupla** es un conjunto ordenado e inmutable de elementos del mismo o diferente tipo.

Tuple items are ordered, unchangeable, and allow duplicate values.

Crear.

Similar a como creamos una lista pero ahora usamos **paréntesis**.

```
>>> empty_tuple = ()

>>> tenerife_geoloc = (28.46824, -16.25462)

>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
```

Truco

Al igual que con las listas, las tuplas admiten diferentes tipos de datos: ('a', 1, True)

Para referirnos a un elemento de una tupla, como en las listas, se usa el operador []:

```
dada la tupla:      t = (1, 2, True, "python")
                    mi_var = t[0]           //mi_var es 1
                    mi_var=t[0:2]          //mi_var es (1,2)
```

Tuplas de un elemento.

Hay que prestar especial atención cuando vamos a crear una **tupla de un único elemento**. La intención primera sería hacerlo de la siguiente manera:

```
>>> one_item_tuple = ('Papá Noel')

>>> one_item_tuple
'Papá Noel'

>>> type(one_item_tuple)
str
```

Realmente, hemos creado una variable de tipo str (cadena de texto). Para crear una tupla de un elemento debemos añadir una **coma** al final:

```
>>> one_item_tuple = ('Papá Noel',)

>>> one_item_tuple
('Papá Noel',)

>>> type(one_item_tuple)
tuple
```

Conversión. Tuple().

Podemos convertir otros tipos de datos en una tupla.

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']

>>> tuple(shopping)
('Agua', 'Aceite', 'Arroz')
```

Esta conversión es válida para aquellos tipos de datos que sean iterables, cadenas de caracteres, listas, diccionarios, conjuntos, etc.

El uso de tuple() sin argumentos equivale a crear una tupla vacía.

```
>>> tuple()
()
```

Operaciones con tuplas.

Con las tuplas podemos realizar todas las operaciones vistas para las listas **salvo las que conlleven una modificación** «in-situ» de la misma:

- reverse()
- append()
- extend()
- remove()
- clear()
- sort()


Truco

Sí es posible aplicar `sorted()` o `reversed()` sobre una tupla ya que no estamos modificando su valor sino creando un nuevo objeto.

Desempaquetado de tuplas.

El **desempaquetado** es una característica de las tuplas que nos permite *asignar una tupla a variables independientes*:

```
>>> a_tuple = ('value1', 'value2', 'value3')
```



```
>>> var1, var2, var3 = a_tuple
```

También es una tupla

```
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')

>>> king1, king2, king3 = three_wise_men

>>> king1
'Melchor'
>>> king2
'Gaspar'
>>> king3
'Baltasar'
```

Intercambio de valores.

A través del desempaquetado de tuplas podemos llevar a cabo el intercambio de los valores de dos variables de manera directa:

```
>>> value1 = 40
>>> value2 = 20

>>> value1, value2 = value2, value1

>>> value1
20
>>> value2
40
```


Nota

A priori puede parecer que esto es algo «natural», pero en la gran mayoría de lenguajes de programación no es posible hacer este intercambio de forma «directa» ya que necesitamos recurrir a una tercera variable «auxiliar» como almacén temporal en el paso intermedio de traspaso de valores.

Listas y Tuplas.

Operaciones comunes:

- `len(sequence)` Returns the length of the sequence
- `for element in sequence` Iterates over each element in the sequence
- `if element in sequence` Checks whether the element is part of the sequence
- `sequence[i]` Accesses the element at index `i` of the sequence, starting at zero
- `sequence[i:j]` Accesses a slice starting at index `i`, ending at index `j-1`. If `i` is omitted, it's 0 by default. If `j` is omitted, it's `len(sequence)` by default.
- `for index, element in enumerate(sequence)` Iterates over both the indexes and the elements in the sequence at the same time

Check out the [official documentation for sequence operations](#).

Listas: operaciones y métodos específicos.

- `list[i] = x` Replaces the element at index `i` with `x`
- `list.append(x)` Inserts `x` at the end of the list
- `list.insert(i, x)` Inserts `x` at index `i`
- `list.pop(i)` Returns the element at index `i`, also removing it from the list. If `i` is omitted, the last element is returned and removed.
- `list.remove(x)` Removes the first occurrence of `x` in the list
- `list.sort()` Sorts the items in the list
- `list.reverse()` Reverses the order of items of the list
- `list.clear()` Removes all the items of the list
- `list.copy()` Creates a copy of the list
- `list.extend(other_list)` Appends all the elements of `other_list` at the end of list

Most of these methods come from the fact that lists are mutable sequences.

For more info, see the [official documentation for mutable sequences](#) and the [list specific documentation](#).

Diccionarios.

En Python un diccionario es también un objeto indexado por **claves** (las palabras) que tienen asociados unos **valores** (los significados).

Los diccionarios en Python tienen las siguientes características:

- Mantienen el **orden** en el que se insertan las claves.
- Son **mutables**, con lo que admiten añadir, borrar y modificar sus elementos.
- Las **claves** deben ser **únicas**. A menudo se utilizan las cadenas de texto como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- Tienen un **acceso muy rápido** a sus elementos, debido a la forma en la que están implementados internamente.

Crear diccionarios.

Para crear un diccionario usamos llaves `{}` rodeando asignaciones **clave: valor** que están separadas por comas.

```

>>> empty_dict = {}

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> population_can = {
...     2015: 2_135_209,
...     2016: 2_154_924,
...     2017: 2_177_048,
...     2018: 2_206_901,
...     2019: 2_220_270
... }

```

Creación con dict().

También es posible utilizar la función dict() para crear diccionarios y no tener que utilizar llaves y comillas.

```

>>> person = dict(
...     name='Guido',
...     surname='Van Rossum',
...     job='Python creator'
... )

>>> person
{'name': 'Guido', 'surname': 'Van Rossum', 'job': 'Python creator'}

```

Conversión.

Para convertir otros tipos de datos en un diccionario podemos usar la función dict():

```
>>> # Diccionario a partir de una lista de cadenas de texto
>>> dict(['a1', 'b2'])
{'a': '1', 'b': '2'}

>>> # Diccionario a partir de una tupla de cadenas de texto
>>> dict(('a1', 'b2'))
{'a': '1', 'b': '2'}

>>> # Diccionario a partir de una lista de listas
>>> dict(['a', 1], ['b', 2])
{'a': 1, 'b': 2}
```

Operaciones con diccionarios.

Obtener un elemento.

Para obtener un elemento de un diccionario basta con escribir la **clave** entre corchetes. [].
si la clave no existe, nos da un error.

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae['anarcoide']
'Que tiende al desorden'
```

Obtener elemento usando get.

Usando get si la clave no existe, devuelve None, salvo que indiquemos otro valor por defecto.


```

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

>>> rae.get('bifronte')
'De dos frentes o dos caras'

>>> rae.get('programación')

>>> rae.get('programación', 'No disponible')
'No disponible'

```

Añadir o modificar un elemento.

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la clave y asignarle un valor:

- Si la clave **ya existía** en el diccionario, **se reemplaza** el valor existente por el nuevo.
- Si la clave **es nueva**, **se añade** al diccionario con su valor. No vamos a obtener un error a diferencia de las listas.

```

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

```

```

>>> rae['enjuiciar'] = 'Someter una cuestión a examen, discusión y juicio'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'}

```

```
>>> rae['enjuiciar'] = 'Instruir, juzgar o sentenciar una causa'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

Creando desde vacío.

Una forma muy habitual de trabajar con diccionarios es utilizar el **patrón creación** partiendo de uno vacío e ir añadiendo elementos poco a poco.

```
>>> VOWELS = 'aeiou'

>>> enum_vowels = {}

>>> for i, vowel in enumerate(VOWELS):
...     enum_vowels[vowel] = i + 1
...

>>> enum_vowels
{'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}
```

Nota

Hemos utilizado la función `enumerate()` que ya vimos para las listas en el apartado: [Iterar usando enumeración](#).

Pertenencia de una clave.

La forma de comprobar la existencia de una clave dentro de un diccionario, es utilizar el operador `in`:

```
>>> 'bifronte' in rae
True

>>> 'almohada' in rae
False

>>> 'montuvio' not in rae
False
```

Obtener todos los elementos.

Python ofrece mecanismos para obtener todos los elementos de un diccionario.

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

Obtener todas **las claves** de un diccionario:

función `keys()`:

```
>>> rae.keys()
dict_keys(['bifronte', 'anarcoide', 'montuvio', 'enjuiciar'])
```

Obtener todos **los valores** de un diccionario:

función `values()`:

```
>>> rae.values()
dict_values([
    'De dos frentes o dos caras',
    'Que tiende al desorden',
    'Campesino de la costa',
    'Instruir, juzgar o sentenciar una causa'
])
```

Obtener todos los **pares «clave-valor»** de un diccionario:

función `items()`:

```
>>> rae.items()
dict_items([
    ('bifronte', 'De dos frentes o dos caras'),
    ('anarcoide', 'Que tiende al desorden'),
    ('montuvio', 'Campesino de la costa'),
    ('enjuiciar', 'Instruir, juzgar o sentenciar una causa')
])
```

Nota

Para este último caso cabe destacar que los «items» se devuelven como una lista de *tuplas*, donde cada tupla tiene dos elementos: el primero representa la clave y el segundo representa el valor.

Iterar sobre un diccionario.

Método básico de iteración sobre un diccionario.

```
user@ubuntu: ~
File Edit View Search Terminal Help
>>> file_counts = {"jpg":10, "txt":14, "csv":2, "py":23}
>>> for extension in file_counts:
...     print(extension)
...
jpg
txt
csv
py
>>>
```

Iterar sobre claves:

```
>>> for word in rae.keys():
...     print(word)
...
bifronte
anarcoide
montuvio
enjuiciar
```

Iterar sobre valores:

```
>>> for meaning in rae.values():  
...     print(meaning)  
...  
De dos frentes o dos caras  
Que tiende al desorden  
Campesino de la costa  
Instruir, juzgar o sentenciar una causa
```

Iterar sobre «clave-valor»:

```
>>> for word, meaning in rae.items():  
...     print(f'{word}: {meaning}')  
...  
bifronte: De dos frentes o dos caras  
anarcoide: Que tiende al desorden  
montuvio: Campesino de la costa  
enjuiciar: Instruir, juzgar o sentenciar una causa
```

[Cuidado con las copias.](#)

Al igual que ocurría con las listas, si hacemos un cambio en un diccionario, se verá reflejado en todas las variables que hagan referencia al mismo. Esto se deriva de su propiedad de ser *mutable*.


```

>>> original_rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> copy_rae = original_rae

>>> original_rae['bifronte'] = 'bla bla bla'

>>> original_rae
{'bifronte': 'bla bla bla',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

>>> copy_rae
{'bifronte': 'bla bla bla',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

```

Una posible solución a este problema es hacer una «copia dura». Para ello Python proporciona la función `copy()`:

```

>>> original_rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> copy_rae = original_rae.copy()

>>> original_rae['bifronte'] = 'bla bla bla'

>>> original_rae
{'bifronte': 'bla bla bla',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

>>> copy_rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}

```

[Borrar elementos.](#)

Por su clave:

Mediante la sentencia [del](#):


```

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> del rae['bifronte']

>>> rae
{'anarcoide': 'Que tiende al desorden', 'montuvio': 'Campesino de la costa'}

```

Por su clave (con extracción):

Mediante la función `pop()` podemos extraer un elemento del diccionario por su clave.

```

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae.pop('anarcoide')
'Que tiende al desorden'

>>> rae
{'bifronte': 'De dos frentes o dos caras', 'montuvio': 'Campesino de la costa'}

>>> rae.pop('bucle')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'bucle'

```

Borrado completo del diccionario:

- Utilizando la función `clear()`:

```

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae.clear()

>>> rae
{}

```

- «Reinicializando» el diccionario a vacío con {}:

```

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae = {}

>>> rae
{}

```

Longitud de un diccionario.

Podemos conocer el número de elementos («clave-valor») que tiene un diccionario con la función `len()`:

```

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

>>> len(rae)
4

```

Combinar diccionarios

Dados dos (o más) diccionarios, es posible «mezclarlos» para obtener una combinación de los mismos. Esta combinación se basa en dos premisas:

- 1.Si la clave no existe, se añade con su valor.
- 2.Si la clave ya existe, se añade con el valor del «último» diccionario en la mezcla.

```
>>> word = 'python'

>>> enough_length = len(word) > 4           # True
>>> right_beginning = word.startswith('p')   # True
>>> min_ys = word.count('y') >= 1           # True

>>> is_cool_word = all([enough_length, right_beginning, min_ys])

>>> if is_cool_word:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Python ofrece dos mecanismos para realizar esta combinación. Vamos a partir de los siguientes diccionarios para ejemplificar su uso:

```
>>> rae1 = {
...     'bifronte': 'De dos frentes o dos caras',
...     'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'
... }

>>> rae2 = {
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa',
...     'enjuiciar': 'Instruir, juzgar o sentenciar una causa'
... }
```

Sin modificar los diccionarios originales:

Mediante el operador **:

```
>>> {**rae1, **rae2}
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

A partir de **Python 3.9** podemos utilizar el operador | para combinar dos diccionarios:

```
>>> rae1 | rae2
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

Modificando los diccionarios originales:

Mediante la función update():

```
>>> rae1.update(rae2)

>>> rae1
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

Diccionarios.

Definición.

`x = {key1:value1, key2:value2}`

Operaciones.

- `len(dictionary)` - Returns the number of items in the dictionary
- `for key in dictionary` - Iterates over each key in the dictionary
- `for key, value in dictionary.items()` - Iterates over each key,value pair in the dictionary
- `if key in dictionary` - Checks whether the key is in the dictionary
- `dictionary[key]` - Accesses the item with key key of the dictionary
- `dictionary[key] = value` - Sets the value associated with key
- `del dictionary[key]` - Removes the item with key key from the dictionary

Métodos.

- `dict.get(key, default)` - Returns the element corresponding to key, or default if it's not present
- `dict.keys()` - Returns a sequence containing the keys in the dictionary
- `dict.values()` - Returns a sequence containing the values in the dictionary
- `dict.update(other_dictionary)` - Updates the dictionary with the items coming from the other dictionary. Existing entries will be replaced; new entries will be added.
- **`dict.clear()` - Removes all the items of the dictionary**

Conjuntos.

Un **conjunto** en Python representa una serie de **valores únicos** y **sin orden establecido**, e **inmutables**.

Creando conjuntos.

Para crear un conjunto basta con separar sus valores por comas y rodearlos de llaves {}.

```
>>> lottery = {21, 10, 46, 29, 31, 94}

>>> lottery
{10, 21, 29, 31, 46, 94}
```

Creando un conjunto vacío. Set().

```
>>> empty_set = set()

>>> empty_set
set()

>>> type(empty_set)
set
```

Conversión.

Para convertir otros tipos de datos en un conjunto podemos usar la función `set()` sobre cualquier iterable:

```
>>> set('aplatanada')
{'a', 'd', 'l', 'n', 'p', 't'}

>>> set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5])
{1, 2, 3, 4, 5}

>>> set(('ADENINA', 'TIMINA', 'TIMINA', 'GUANINA', 'ADENINA', 'CITOSINA'))
{'ADENINA', 'CITOSINA', 'GUANINA', 'TIMINA'}

>>> set({'manzana': 'rojo', 'plátano': 'amarillo', 'kiwi': 'verde'})
{'kiwi', 'manzana', 'plátano'}
```

Operaciones con conjuntos

Obtener un elemento

En un conjunto no existe un orden establecido para sus elementos, por lo tanto **no podemos acceder a un elemento en concreto**.

De este hecho se deriva igualmente que **no podemos modificar un elemento existente**, ya que ni siquiera tenemos acceso al mismo. Python sí nos permite añadir o borrar elementos de un conjunto.

Añadir un elemento

Para añadir un elemento a un conjunto debemos utilizar la [función add\(\)](#). Como ya hemos indicado, al no importar el orden dentro del conjunto, la inserción no establece a priori la posición donde se realizará.

A modo de ejemplo, vamos a partir de un conjunto que representa a los cuatro integrantes originales de *The Beatles*. Luego añadiremos a un nuevo componente:

```
>>> # John Lennon, Paul McCartney, George Harrison y Ringo Starr
>>> beatles = set(['Lennon', 'McCartney', 'Harrison', 'Starr'])

>>> beatles.add('Best') # Pete Best

>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

Obtener un elemento.

En un conjunto no existe un orden establecido para sus elementos, por lo tanto **no podemos acceder a un elemento en concreto**.

De este hecho se deriva igualmente que **no podemos modificar un elemento existente**, ya que ni siquiera tenemos acceso al mismo. Python sí nos permite añadir o borrar elementos de un conjunto.

Borrar elementos

Para borrar un elemento de un conjunto podemos utilizar la función `remove()`.

```
>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> beatles.remove('Best')

>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

Longitud de un conjunto. `len()`:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> len(beatles)
4
```

Iterar sobre un conjunto.

Tal y como hemos visto para otros tipos de datos *iterables*, la forma de recorrer los elementos de un conjunto es utilizar la sentencia `for`:

```
>>> for beatle in beatles:
...     print(beatle)
...
Harrison
McCartney
Starr
Lennon
```

Pertenencia de elemento.

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> 'Lennon' in beatles
True

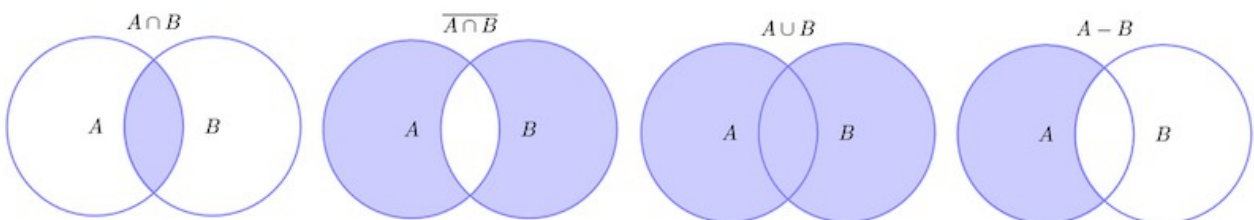
>>> 'Fari' in beatles
False
```

Teoría de conjuntos

Vamos a partir de dos conjuntos $A=\{1,2\}$ y $B=\{2,3\}$ para ejemplificar las distintas operaciones que se pueden hacer entre ellos basadas en los [Diagramas de Venn](#) y la [Teoría de Conjuntos](#):

```
>>> A = {1, 2}

>>> B = {2, 3}
```



Diagramas de Venn

Intersección

$A \cap B$ – Elementos que están a la vez en A y en B :

```
>>> A & B
{2}

>>> A.intersection(B)
{2}
```

Unión

$A \cup B$ – Elementos que están tanto en A como en B :

```
>>> A | B
{1, 2, 3}

>>> A.union(B)
{1, 2, 3}
```

Diferencia

$A - B$ – Elementos que están en A y no están en B :

```
>>> A - B
{1}

>>> A.difference(B)
{1}
```

Diferencia simétrica

$\overline{A \cap B}$ – Elementos que están en A o en B pero no en ambos conjuntos:

```
>>> A ^ B
{1, 3}

>>> A.symmetric_difference(B)
{1, 3}
```

Inclusión

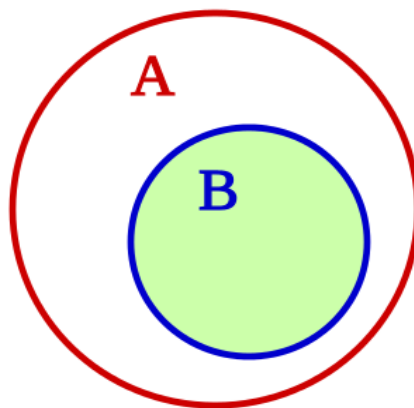
Un conjunto B es un **subconjunto** de otro conjunto A si todos los elementos de B están incluidos en A .

- Un conjunto A es un **superconjunto** de otro conjunto B si todos los elementos de B están incluidos en A .

•

Veamos un ejemplo con los siguientes conjuntos:

```
>>> A = {2, 4, 6, 8, 10}
>>> B = {4, 6, 8}
```



Subconjuntos y Superconjuntos

En Python podemos realizar comprobaciones de inclusión (subconjuntos y superconjuntos) utilizando operadores clásicos de comparación:

$$B \subset A$$

```
>>> B < A # subconjunto  
True
```

$$B \subseteq A$$

```
>>> B <= A  
True
```

$$A \supset B$$

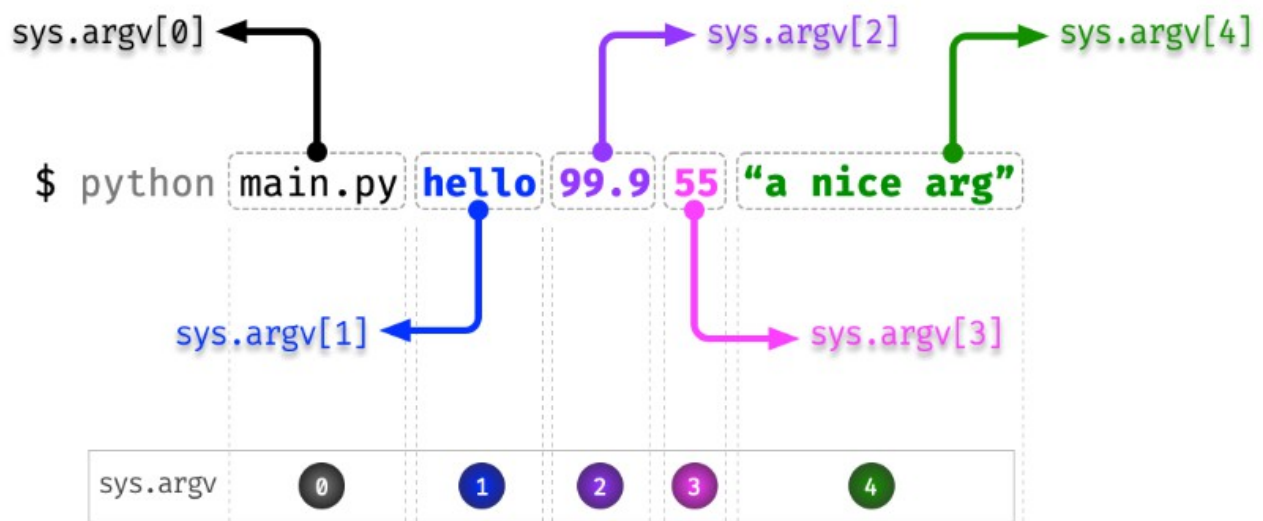
```
>>> A > B # superconjunto  
True
```

$$A \supseteq B$$

```
>>> B >= A  
True
```

sys.argv

Cuando queramos ejecutar un programa Python desde **línea de comandos**, tendremos la posibilidad de acceder a los argumentos de dicho programa. Para ello se utiliza una lista que la encontramos dentro del módulo sys y que se denomina argv:



Acceso a parámetros en línea de comandos

Veamos una aplicación de lo anterior en un programa que convierte un número decimal a una determinada base, ambos argumentos pasados por línea de comandos:

```
1 import sys
2
3 number = int(sys.argv[1])
4 tobase = int(sys.argv[2])
5
6 match tobase:
7     case 2:
8         result = f'{number:b}'
9     case 8:
10        result = f'{number:o}'
11    case 16:
12        result = f'{number:x}'
13    case _:
14        result = None
15
16 if result is None:
17     print(f'Base {tobase} not implemented!')
18 else:
19     print(result)
```


Si lo ejecutamos obtenemos lo siguiente:

```
$ python dec2base.py 65535 2
111111111111111111
```

Funciones matemáticas

Python nos ofrece, entre otras, estas tres funciones matemáticas básicas que se pueden aplicar sobre listas.

Suma de todos los valores:

Mediante la función `sum()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> sum(data)
28
```

Mínimo de todos los valores:

Mediante la función `min()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> min(data)
1
```

Máximo de todos los valores:

Mediante la función `max()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> max(data)
9
```

POO en Python.

Los conceptos son modelados como clases y objetos. Una idea es definida usando una clase y una instancia de esta clase es llamada objeto. Todo en Python es un objeto, incluidas las listas, Strings, diccionarios, números....

Primer Ejemplo.

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4     def greeting(self):
5         # Should return "hi, my name is " followed by the name of the Per
6         return "hi, my name is " + self.name
7
8 # Create a new instance with a name of your choice
9 some_person = Person("Maria")
10 # Call the greeting method
11 print(some_person.greeting())
12
```

Ejecutar

Restablecer

Las clases además tienen atributos y métodos asociados a ellas, Los **atributos** son las características de la clase, y los **métodos** son funciones que son parte de la clase.

Crear una clase

Para crear una clase usamos **class** seguido por el nombre de la clase y dos puntos, se recomienda que las clases comiencen con mayúsculas.

Ej. creamos una clase llamada MyClass, con una propiedad llamada x:

```
class MyClass:
    x = 5
```

Crear clase vacía

```
File Edit View Search Terminal Help
>>> class Apple:
...     pass
...
>>>
```

Usamos **pass** para indicar que por ahora la clase está vacía.

Definir clase con atributos:

```
1 >>> class Apple:
2 ...     color = ""
3 ...     flavor = ""
4 ...
```

Podemos crear una instancia de la clase asignándola a una variable, esto se hace llamando al nombre de la clase como si fuese una función. Podemos establecer los atributos de nuestra instancia de la clase usando la notación punto.

La notación punto es usada para establecer o recuperar los atributos del objeto, así como llamar a los métodos asociados con la clase.

```
1 >>> jonagold = Apple()
2 >>> jonagold.color = "red"
3 >>> jonagold.flavor = "sweet"
```

Hemos creado una instancia de Apple llamada Jonagold y establecido sus atributos color y flavor para este objeto Apple, podemos crear otra instancia de la clase Apple y establecer diferentes atributos para diferenciar entre las dos diferentes variedades de Apple.

```
1 >>> golden = Apple()
2 >>> golden.color = "Yellow"
3 >>> golden.flavor = "Soft"
```

Crear un objeto.

Ej 1: Usamos la clase MyClass para crear un objeto. Creamos un objeto llamado p1, e imprimimos el valor de x.

```
p1 = MyClass()
print(p1.x)
```

Ej 2: Creamos una instancia de la clase Apple y asignamos valores para después imprimir los valores de los atributos.

```
>>> class Apple:
...     color = ""
...     flavor = ""
...
>>> jonagold = Apple()
>>> jonagold.color = "red"
>>> jonagold.flavor = "sweet"
>>> print(jonagold.color)
red
>>> print(jonagold.flavor)
sweet
>>> █
```

importante:

```
>>> print(jonagold.color.upper())
RED
>>> █
```

Función `__init__()`

Todas las clases tienen una función llamada `__init__()`, la cual es siempre ejecutada cuando la clase es iniciada.

Usamos `__init__()` para asignar valores a las propiedades de los objetos, u otras operaciones que son necesarias.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Nota: `__init__()` es llamada automáticamente cada vez es usada para crear un nuevo objeto.

En lugar de crear una clase con valores vacío o valores por defecto, podemos establecer estos valores cuando creamos una instancia.

Con esto nos aseguramos de no olvidar valores importantes, para hacer esto usamos un método llamado **constructor**. Ejemplo de Apple class con un método constructor definido.

```
1 >>> class Apple:
2 ...     def __init__(self, color, flavor):
3 ...         self.color = color
4 ...         self.flavor = flavor
```

Cuando llamamos al nombre de la clase, el constructor es llamado. Este constructor se llama siempre `__init__`.

Debes recordar que los métodos especiales comienzan y terminan con dos guiones bajos.

```
1 >>> jonagold = Apple("red", "sweet")
2 >>> print(jonagold.color)
3 Red
```

Función `__str__()`

Otro metodo especial `__str__` nos permite definir como una instancia de un objeto es mostrado cuando es llamada la funcion `print()`.

Si un objeto no tiene este método definido usa la representación que imprime la posición del objeto en memoria.

```
1 >>> class Apple:
2 ...     def __init__(self, color, flavor):
3 ...         self.color = color
4 ...         self.flavor = flavor
5 ...     def __str__(self):
6 ...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)
7 ...
```

```
1 >>> jonagold = Apple("red", "sweet")
2 >>> print(jonagold)
3 This apple is red and its flavor is sweet
```

Ejemplo de representation de un objeto SIN la función `__str__()`:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1)
```

```
<__main__.Person object at 0x15039e602100>
```

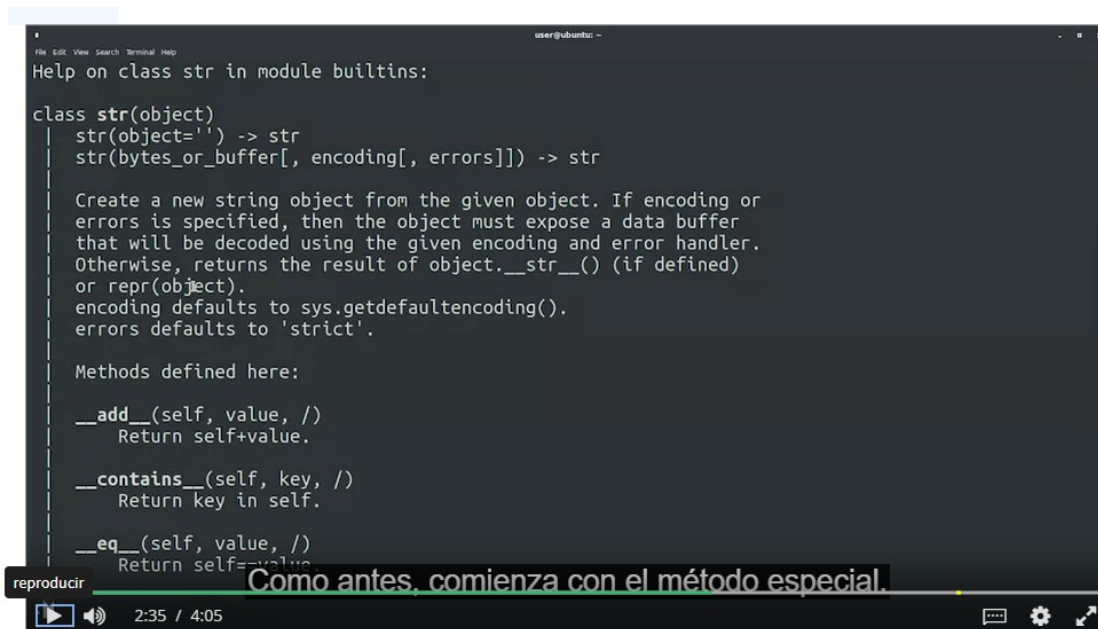
Documentación de clases, métodos y funciones.

- Podemos añadir documentación usando docstrings después de la definición.

```
1 class ClassName:
2     """Documentation for the class."""
3     def method_name(self, other_parameters):
4         """Documentation for the method."""
5         body_of_method
6
7     def function_name(parameters):
8         """Documentation for the function."""
9         body_of_function
10
```

Usamos la función `dir` para mostrar una lista de metodos de una clase.

Usamos la función `help` para ver la documentación de una clase.



```
user@ubuntu: ~
Help on class str in module builtins:

class str(object)
| str(object='') -> str
| str(bytes_or_buffer[, encoding[, errors]]) -> str
|
| Create a new string object from the given object. If encoding or
| errors is specified, then the object must expose a data buffer
| that will be decoded using the given encoding and error handler.
| Otherwise, returns the result of object.__str__() (if defined)
| or repr(object).
| encoding defaults to sys.getdefaultencoding().
| errors defaults to 'strict'.
|
| Methods defined here:
|
| __add__(self, value, /)
|     Return self+value.
|
| __contains__(self, key, /)
|     Return key in self.
|
| __eq__(self, value, /)
|     Return self==value.
|
| ...
```

Usamos la función `type()` para ver a que clase pertenece la variable.

Ejemplos.

```
1  # "If you have an apple and I have an apple and we exchange these apples then
2  # you and I will still each have one apple. But if you have an idea and I have
3  # an idea and we exchange these ideas, then each of us will have two ideas."
4  # George Bernard Shaw
5
6  class Person:
7      apples = 0
8      ideas = 0
9
10     johanna = Person()
11     johanna.apples = 1
12     johanna.ideas = 1
13
14     martin = Person()
15     martin.apples = 2
16     martin.ideas = 1
17
18     def exchange_apples(you, me):
19         #Here, despite G.B. Shaw's quote, our characters have started with      #different amo
20         #We're going to have Martin and Johanna exchange ALL their apples with #one another.
21         #Hint: how would you switch values of variables,
22         #so that "you" and "me" will exchange ALL their apples with one another?
23         #Do you need a temporary variable to store one of the values?
24         #You may need more than one line of code to do that, which is OK.
25         vble_aux = you.apples
26
27         you.apples = me.apples
28         me.apples = vble_aux
29
30         return you.apples, me.apples
31
32     def exchange_ideas(you, me):
33         # "you" and "me" will share our ideas with one another.
34         #What operations need to be performed, so that each object receives
35         #the shared number of ideas?
36         #Hint: how would you assign the total number of ideas to
37         #each idea attribute? Do you need a temporary variable to store
38         #the sum of ideas, or can you find another way?
39         #Use as many lines of code as you need here.
40         vbleIdeas_aux = you.ideas
```

[Ejecutar](#)[Restablecer](#)

Johanna has 2 apples and Martin has 1 apples
Johanna has 2 ideas and Martin has 2 ideas


```
51
52  def exchange_ideas(you, me):
53      # "you" and "me" will share our ideas with one another.
54      # What operations need to be performed, so that each object receives
55      # the shared number of ideas?
56      # Hint: how would you assign the total number of ideas to
57      # each idea attribute? Do you need a temporary variable to store
58      # the sum of ideas, or can you find another way?
59      # Use as many lines of code as you need here.
60      vbleIdeas_aux = you.ideas
61      you.ideas = you.ideas + me.ideas
62      me.ideas = me.ideas + vbleIdeas_aux
63      return you.ideas, me.ideas
64
65 exchange_apples(johanna, martin)
66 print("Johanna has {} apples and Martin has {} apples".format(johanna.apples, martin.apples))
67 exchange_ideas(johanna, martin)
68 print("Johanna has {} ideas and Martin has {} ideas".format(johanna.ideas, martin.ideas))
69
70
71
72
```

Ejecutar

Restablecer

Johanna has 2 apples and Martin has 1 apples
Johanna has 2 ideas and Martin has 2 ideas

- ```

1 class Furniture:
2 color = ""
3 material = ""
4
5 table = Furniture()
6 table.color = "brown"
7 table.material = "wood"
8
9 couch = Furniture()
10 couch.color = "red"
11 couch.material = "leather"
12
13 def describe_furniture(piece):
14 return ("This piece of furniture is made of {} {}".format(piece.color, piece.material))
15
16 print(describe_furniture(table))
17 # Should be "This piece of furniture is made of brown wood"
18 print(describe_furniture(couch))
19 # Should be "This piece of furniture is made of red leather"

```
- Ejecutar
- Restablecer
- ```

This piece of furniture is made of brown wood
This piece of furniture is made of red leather

```

 **Correcto**

Right on! You're working well with classes, objects, and instances!

FIN EJEMPLOSSSSSSSSSSSSSSSSSSSSSSSS – falta añadir más ejemplos, pendiente de actualizar.-

Object Methods

Ejemplo.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

`self` parametro es una referencia de la instancia de la clase y es usada para acceder a la variable que pertenece a la clase.

Modificar propiedades de objetos.

```
p1.age = 40
```

Borrar propiedades de objetos. Del.

```
del p1.age
```

Borrar objetos.Del.

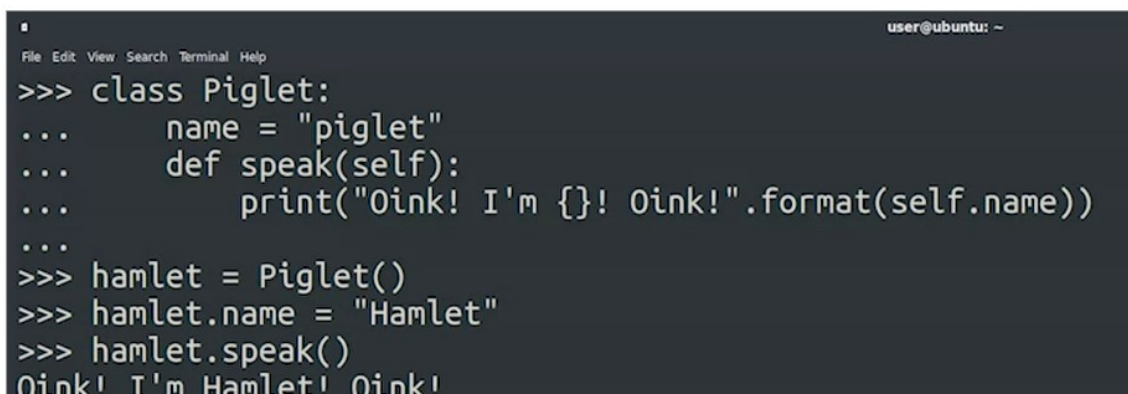
```
del p1
```

sentencia pass

La definición de clase no debe quedar vacía, pero si queremos hacerlo usamos la sentencia **pass**.

```
class Person:  
    pass
```

Instance method:



```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> class Piglet:  
...     name = "piglet"  
...     def speak(self):  
...         print("Oink! I'm {}! Oink!".format(self.name))  
...  
>>> hamlet = Piglet()  
>>> hamlet.name = "Hamlet"  
>>> hamlet.speak()  
Oink! I'm Hamlet! Oink!
```

Esta vez hemos estudiado el cuerpo de la clase definiendo un atributo llamado nombre con un valor predeterminado de Piglet.

Podemos cambiar ese valor más tarde, pero es una buena idea configurarlo ahora para asegurarse de que nuestra variable se inicializa. Si miras detenidamente cómo escribimos el método de newspeak, verás que está usando el valor de `self.name` para saber qué nombre imprimir.

Esto significa que está accediendo al nombre del atributo desde la instancia actual de Piglet.

Variables that have different values for different instances of the same class are called **instance variables**.

Métodos que devuelven valores:

```
File Edit View Search Terminal Help
>>> class Piglet:
...     years = 0
...     def pig_years(self):
...         return self.years * 18
...
>>> piggy = Piglet()
>>> print(piggy.pig_years())
0
>>> piggy.years = 2
>>> print(piggy.pig_years())
36
>>>
```

Constructors and Other Special Methods

Así que vamos a establecer esos valores a medida que creamos la instancia.

De esta manera, sabemos que nuestra instancia tiene todos los valores importantes en ella desde el momento en que se crea y no tenemos que preocuparnos por ello.

Para hacer esto, necesitamos usar un método especial llamado constructor.

Volvamos a nuestro ejemplo de manzana para ver esto en acción.

El constructor de la clase es el método que se llama cuando se llama al nombre de la clase.

Siempre se llama `__init__`. Es posible que recuerde que todos los métodos que comienzan y terminan con dos guiones bajos son métodos especiales.

Aquí, hemos definido un constructor, un método especial muy importante. Este método en la parte superior de la variable `self` que representa la instancia recibe dos parámetros más: `color` y `sabor`. A continuación, el método establece esos valores como los valores de la instancia actual.

```
File Edit View Search Terminal Help
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...
>>> 
```

```
File Edit View Search Terminal Help
>>> class Apple:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...
>>> jonagold = Apple("red", "sweet")
>>> print(jonagold.color)
red
>>> 
```

Herencia en Python.

La herencia nos permite definir una clase que hereda todos los metodos y propiedades de otra clase.

Parent class La clase de la cual se hereda, también llamada clase base.

Child class es la clase que hereda de otra clase, también llamada clase derivada.

Crear clase padre

Cualquier clase puede ser una clase padre, la sintaxis es la misma para la creación de cualquier clase.

ejemplo.

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Crear clase hijo

Para crear una clase que herede la funcionalidad de otra clase, enviamos la clase padre como parámetro cuando se crea la clase hijo.

ejemplo:

crear la clase **Student**, que hereda de la clase **Person**:

```
class Student(Person):  
    pass
```

Ahora la clase Student tiene las mismas propiedades y métodos que la clase Person.

Ejemplo:

Usar la clase Student para crear un objeto, y ejecutar el metodo printname:

```
x = Student("Mike", "Olsen")  
x.printname()
```

Función __init__()

Al igual que con cualquier otra clase, añadimos la funcion __init__() a la clase hijo, pero en este caso la función init añadida sobrescribe la heredada del padre.

Ej: Add the __init__() function to the Student class:

```
class Student(Person):  
    def __init__(self, fname, lname):  
        #add properties etc.
```

Para mantener la clase init del padre, añadimos una llamada a la función de la clase padre, ej:

```
class Student(Person):  
    def __init__(self, fname, lname):  
        Person.__init__(self, fname, lname)
```


Función super()

Python tiene la función `super()` que hará que la clase hijo herede todos los metodos y propiedades de su padre.

```
class Student(Person):  
    def __init__(self, fname, lname):  
        super().__init__(fname, lname)
```

Para usar la función `super()`, no usamos el nombre del padre, automaticamente se hereda los metodos y propiedades.

Añadir propiedades.

Ejemplo: añadir la propiedad `graduationyear`.

```
class Student(Person):  
    def __init__(self, fname, lname):  
        super().__init__(fname, lname)  
        self.graduationyear = 2019
```

En el ejemplo anterior, el año 2019 debería ser una variable pasada a la clase Student cuando creamos el objeto. Para hacer esto se añade otro parámetro en `__init__()` :

EJ:

```
class Student(Person):  
    def __init__(self, fname, lname, year):  
        super().__init__(fname, lname)  
        self.graduationyear = year  
  
x = Student("Mike", "Olsen", 2019)
```

Añadir métodos.

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

Si se añade un método en la clase hijo con el mismo nombre que la función en la clase padre, la herencia de la clase padre será sobrescrita.

Ejemplo 1.

```
File Edit View Search Terminal Help
>>> class Fruit:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...
>>> class Apple(Fruit):
...     pass
...
>>> class Grape(Fruit):
...     pass
...
>>>
```

```
File Edit View Search Terminal Help
>>> class Fruit:
...     def __init__(self, color, flavor):
...         self.color = color
...         self.flavor = flavor
...
>>> class Apple(Fruit):
...     pass
...
>>> class Grape(Fruit):
...     pass
...
>>> granny_smith = Apple("green", "tart")
>>> carnelian = Grape("purple", "sweet")
>>> print(granny_smith.flavor)
tart
>>> print(carnelian.color)
purple
>>>
```

Ejemplo 2.

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> class Animal:  
...     sound = ""  
...     def __init__(self, name):  
...         self.name = name  
...     def speak(self):  
...         print("{sound} I'm {name}! {sound}".format(  
...             name=self.name, sound=self.sound))  
...  
>>> class Piglet(Animal):  
...     sound = "Oink!"  
...  
>>> █
```

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> class Animal:  
...     sound = ""  
...     def __init__(self, name):  
...         self.name = name  
...     def speak(self):  
...         print("{sound} I'm {name}! {sound}".format(  
...             name=self.name, sound=self.sound))  
...  
>>> class Piglet(Animal):  
...     sound = "Oink!"  
...  
>>> hamlet = Piglet("Hamlet")  
>>> hamlet.speak()  
Oink! I'm Hamlet! Oink!  
>>> █
```

```
user@ubuntu: ~  
File Edit View Search Terminal Help  
>>> class Animal:  
...     sound = ""  
...     def __init__(self, name):  
...         self.name = name  
...     def speak(self):  
...         print("{sound} I'm {name}! {sound}".format(  
...             name=self.name, sound=self.sound))  
...  
>>> class Piglet(Animal):  
...     sound = "Oink!"  
...  
>>> hamlet = Piglet("Hamlet")  
>>> hamlet.speak()  
Oink! I'm Hamlet! Oink!  
>>> class Cow(Animal):  
...     sound = "Mooooo"  
...  
>>> milky = Cow("Milky White")  
>>> milky.speak()  
Mooooo I'm Milky White! Mooooo
```

Ejemplo 3.

```
1  class Clothing:  
2      material = ""  
3      def __init__(self, name):  
4          self.name = name  
5      def checkmaterial(self):  
6          print("This {} is made of {}".format(self.name, self.material))  
7  
8  class Shirt(Clothing):  
9      material = "Cotton"  
10  
11  polo = Shirt("Polo")  
12  polo.checkmaterial()
```

Ejecutar

Restablecer

Let's expand a bit on our Clothing classes from the previous in-video question. Your mission: Finish the "Stock_by_Material" method and iterate over the amount of each item of a given material that is in stock. When you're finished, the script should add up to 10 cotton Polo shirts.

```
12     n=0
13     for item in Clothing.stock['material']:
14         if item == material:
15             count += Clothing.stock['amount'][n]
16             n+=1
17     return count
18
19 class shirt(Clothing):
20     material="Cotton"
21 class pants(Clothing):
22     material="Cotton"
23
24 polo = shirt("Polo")
25 sweatpants = pants("Sweatpants")
26 polo.add_item(polo.name, polo.material, 4)
27 sweatpants.add_item(sweatpants.name, sweatpants.material, 6)
28 current_stock = polo.Stock_by_Material("Cotton")
29 print(current_stock)
```

Ejecutar

Restablecer

Here is your output:

```
10
```

Nice job! You successfully used composition to reuse the Clothing.stock attribute and stock_by_material() function of the Clothing class to take stock of the Cotton shirts!

Let's expand a bit on our Clothing classes from the previous in-video question. Your mission: Finish the "Stock_by_Material" method and iterate over the amount of each item of a given material that is in stock. When you're finished, the script should add up to 10 cotton Polo shirts.

```
1 class Clothing:
2     stock={ 'name': [], 'material' :[], 'amount':[]}
3     def __init__(self,name):
4         material = ""
5         self.name = name
6     def add_item(self, name, material, amount):
7         Clothing.stock['name'].append(self.name)
8         Clothing.stock['material'].append(self.material)
9         Clothing.stock['amount'].append(amount)
10    def Stock_by_Material(self, material):
11        count=0
```

Object Composition

Podemos tener la situación que dos clases diferentes están relacionadas pero no existe herencia. Esto se denomina composición, cuando una clase hace uso del código contenido en otra clase.

Por ejemplo, imagina que tenemos una clase `Package` que representa un paquete de software, contiene atributos sobre el paquete de software como el nombre, versión y tamaño, además tenemos una clase `Repository` que representa todos los paquetes disponibles para la instalación. No existe relación de herencia entre ellas, La clase `Repository` contiene un diccionario o lista de paquetes.

Ejemplo.

```
1  >>> class Repository:
2  ...     def __init__(self):
3  ...         self.packages = {}
4  ...     def add_package(self, package):
5  ...         self.packages[package.name] = package
6  ...     def total_size(self):
7  ...         result = 0
8  ...         for package in self.packages.values():
9  ...             result += package.size
10 ...         return result
```

En el método constructor inicializamos el diccionario de paquetes que contendrá los paquetes disponibles en el repositorio. Inicializamos el diccionario en el constructor para asegurarnos que cada instancia de la clase `Repository` tiene su propio diccionario.

Entonces definimos el método `add_package` method, que toma como parámetros un objeto `Package` y entonces se añade a nuestro diccionario, usando el nombre del atributo `package` como key.

Finalmente definimos el método `total_size` que cuenta el tamaño total de todos los paquetes contenidos en el repositorio.

Necesitamos iterar sobre los paquetes contenidos en el diccionario, sumando todos sus tamaños.

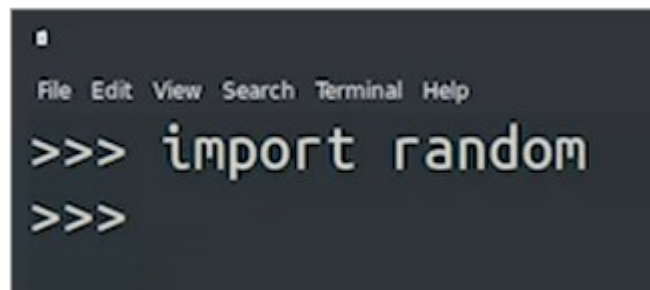
Errata, `for package in self.packages.values():` #falta la s

Python Modules.

Modules

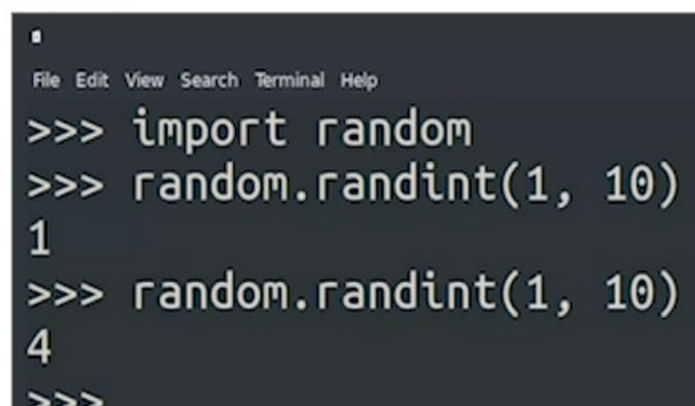
Used to organize functions, classes, and other data together in a structured way

Usamos import para poder importar un modulo.



```
File Edit View Search Terminal Help
>>> import random
>>>
```

Ahora podemos usar la funcion randint del modulo random



```
File Edit View Search Terminal Help
>>> import random
>>> random.randint(1, 10)
1
>>> random.randint(1, 10)
4
>>>
```

```
File Edit View Search Terminal Help
>>> import datetime
>>> now = datetime.datetime.now()
>>> type(now)
<class 'datetime.datetime'>
>>> print(now)
2020-01-04 13:54:00.056680
>>>
```

```
File Edit View Search Terminal Help
>>> import datetime
>>> now = datetime.datetime.now()
>>> type(now)
<class 'datetime.datetime'>
>>> print(now)
2020-01-04 13:54:00.056680
>>> now.year
2020
>>> print(now + datetime.timedelta(days=28))
2020-02-01 13:54:00.056680
>>> █
```


Augmenting Python with Modules

Python modules are separate files that contain classes, functions, and other data that allow us to import and make use of these methods and classes in our own code. Python comes with a lot of modules out of the box. These modules are referred to as the Python Standard Library. You can make use of these modules by using the **import** keyword, followed by the module name. For example, we'll import the **random** module, and then call the **randint** function within this module:

```
1 >>> import random
2 >>> random.randint(1,10)
3 8
4 >>> random.randint(1,10)
5 7
6 >>> random.randint(1,10)
7 1
```

This function takes two integer parameters and returns a random integer between the values we pass it; in this case, 1 and 10. You might notice that calling functions in a module is very similar to calling methods in a class. We use dot notation here too, with a period between the module and function names.

Let's take a look at another module: **datetime**. This module is super helpful when working with dates and times.

```
1 >>> import datetime
2 >>> now = datetime.datetime.now()
3 >>> type(now)
4 <class 'datetime.datetime'>
5 >>> print(now)
6 2019-04-24 16:54:55.155199
```

First, we import the module. Next, we call the **now()** method which belongs to the **datetime** class contained within the **datetime** module. This method generates an instance of the datetime class for the current date and time. This instance has some methods which we can call:

```
1 >>> print(now)
2 2019-04-24 16:54:55.155199
3 >>> now.year
4 2019
5 >>> print(now + datetime.timedelta(days=28))
6 2019-05-22 16:54:55.155199
```

When we call the `print` function with an instance of the `datetime` class, we get the date and time printed in a specific format. This is because the `datetime` class has a `__str__` method defined which generates the formatted string we see here. We can also directly call attributes and methods of the class, as with `now.year` which returns the year attribute of the instance.

Lastly, we can access other classes contained in the `datetime` module, like the `timedelta` class. In this example, we're creating an instance of the `timedelta` class with the parameter of 28 days. We're then adding this object to our instance of the `datetime` class from earlier and printing the result. This has the effect of adding 28 days to our original `datetime` object.

Supplemental Reading for Code Reuse

The official Python documentation lists all the modules included in the standard library. It even has a turtle in it...

[Pypi](https://pypi.org/) is the Python repository and index of an impressive number of modules developed by Python programmers around the world.

<https://pypi.org/>

Ejemplos Herencia y composición.

Code Reuse

Let's put what we learned about code reuse all together.

First, let's look back at **inheritance**. Run the following cell that defines a generic `Animal` class.

```
In [8]: class Animal:
        name = ""
        category = ""

        def __init__(self, name):
            self.name = name

        def set_category(self, category):
            category = category
```

What we have is not enough to do much -- yet. That's where you come in.

In the next cell, define a `Turtle` class that inherits from the `Animal` class. Then go ahead and set its category. For instance, a turtle is generally considered a reptile. Although modern cladistics call this categorization into question, for purposes of this exercise we will say turtles are reptiles!

```
In [9]: class Turtle(Animal):
        category = "reptile"
```

Run the following cell to check whether you correctly defined your `Turtle` class and set its category to reptile.

```
In [10]: print(Turtle.category)

reptile
```

Was the output of the above cell reptile? If not, go back and edit your `Turtle` class making sure that it inherits from the `Animal` class and its category is properly set to reptile. Be sure to re-run that cell once you've finished your edits. Did you get it? If so, great!

Next, let's practice **composition** a little bit. This one will require a second type of `Animal` that is in the same category as the first. For example, since you already created a `Turtle` class, go ahead and create a `Snake` class. Don't forget that it also inherits from the `Animal` class and that its category should be set to `reptile`.

```
class Snake(Animal):  
    category = "reptile"
```

Now, let's say we have a large variety of `Animal`s (such as turtles and snakes) in a Zoo. Below we have the `Zoo` class. We're going to use it to organize our various `Animal`s. Remember, inheritance says a `Turtle` is an `Animal`, but a `Zoo` is not an `Animal` and an `Animal` is not a `Zoo` -- though they are related to one another.

Fill in the blanks of the `Zoo` class below so that you can use `zoo.add_animal()` to add instances of the `Animal` subclasses you created above. Once you've added them all, you should be able to use `zoo.total_of_category()` to tell you exactly how many individual `Animal` types the `Zoo` has for each category! Be sure to run the cell once you've finished your edits.

```
class Zoo:  
    def __init__(self):  
        self.current_animals = {}  
  
    def add_animal(self, animal):  
        self.current_animals[animal.name] = animal.category  
  
    def total_of_category(self, category):  
        result = 0  
        for animal in self.current_animals.values():  
            if animal == category:  
                result += 1  
  
        return result  
  
zoo = Zoo()
```

Run the following cell to check whether you properly filled in the blanks of your `Zoo` class.

```
turtle = Turtle("Turtle") #create an instance of the Turtle class  
snake = Snake("Snake") #create an instance of the Snake class  
  
zoo.add_animal(turtle)  
zoo.add_animal(snake)  
  
print(zoo.total_of_category("reptile")) #how many zoo animal types in the reptile category
```

2

Was the output of the above cell 2? If not, go back and edit the `Zoo` class making sure to fill in the blanks with the appropriate attributes. Be sure to re-run that cell once you've finished your edits.

Did you get it? If so, perfect! You have successfully defined your `Turtle` and `Snake` subclasses as well as your `Zoo` class. You are all done with this notebook. Great work!

otro ejercicio:

Assessment - Object-oriented programming

In this exercise, we'll create a few classes to simulate a server that's taking connections from the outside and then a load balancer that ensures that there are enough servers to serve those connections.

To represent the servers that are taking care of the connections, we'll use a `Server` class. Each connection is represented by an id, that could, for example, be the IP address of the computer connecting to the server. For our simulation, each connection creates a random amount of load in the server, between 1 and 10.

Run the following code that defines this `Server` class.

```
#Begin Portion 1#
import random

class Server:
    def __init__(self):
        """Creates a new server instance, with no active connections."""
        self.connections = {}

    def add_connection(self, connection_id):
        """Adds a new connection to this server."""
        connection_load = random.random()*10+1
        # Add the connection to the dictionary with the calculated load
        self.connections[connection_id] = connection_load

    def close_connection(self, connection_id):
        """Closes a connection on this server."""
        # Remove the connection from the dictionary
        for conection in self.connections:
            if conection == connection_id:
                self.connections[conection] = 0

    def load(self):
        """Calculates the current load for all connections."""
        total = 0
        # Add up the load for each of the connections
        for conection in self.connections.values():
            total += conection
        return total

    def __str__(self):
        """Returns a string with the current load of the server"""
        return "{:.2f}%".format(self.load())

#End Portion 1#
```

Now run the following cell to create a `Server` instance and add a connection to it, then check the load:

Now run the following cell to create a Server instance and add a connection to it, then check the load:

```
server = Server()
server.add_connection("192.168.1.1")

print(server.load())
```

3.2869954392967706

After running the above code cell, if you get a **NameError** message, be sure to run the Server class definition code block first.

The output should be 0. This is because some things are missing from the Server class. So, you'll need to go back and fill in the blanks to make it behave properly.

Go back to the Server class definition and fill in the missing parts for the `add_connection` and `load` methods to make the cell above print a number different than zero. As the load is calculated randomly, this number should be different each time the code is executed.

Hint: Recall that you can iterate through the values of your connections dictionary just as you would any sequence.

Great! If your output is a random number between 1 and 10, you have successfully coded the `add_connection` and `load` methods of the Server class. Well done!

What about closing a connection? Right now the `close_connection` method doesn't do anything. Go back to the Server class definition and fill in the missing code for the `close_connection` method to make the following code work correctly:

```
server.close_connection("192.168.1.1")
print(server.load())
```

0

You have successfully coded the `close_connection` method if the cell above prints 0.

Hint: Remember that `del dictionary[key]` removes the item with key `key` from the dictionary.

