

Herramienta de trabajo Google Colab (Google Colaboratory)

Google Colab es una herramienta para escribir y ejecutar código Python en la nube de Google.

También es posible incluir texto enriquecido, enlaces e imágenes. En caso de necesitar altas prestaciones de cómputo, el entorno permite configurar algunas propiedades del equipo sobre el que se ejecuta el código.

En definitiva, el uso de **Google Colab** permite disponer de un entorno para llevar a cabo tareas que serían difíciles de realizar en un equipo personal. Por otro lado, siguiendo la idea de **Google Drive**, **Google Colab** brinda la opción de compartir los códigos realizados lo que es ideal para trabajos en equipo.

Herramienta de trabajo Google Colab (Google Colaboratory)

Google es bastante agresivo en la investigación de IA. Durante muchos años, desarrolló un marco de inteligencia artificial llamado **TensorFlow** y una herramienta de desarrollo llamada Colaboratory. Hoy TensorFlow es de código abierto y, desde 2017, Google hizo que Colaboratory fuera gratuito para uso público. Colaboratory ahora se conoce como Google Colab o simplemente Colab.

Otra característica atractiva que ofrece Google a los desarrolladores es el uso de GPU y TPU.

Una **CPU**, o Unidad Central de Procesamiento, es un procesador de propósito general.

Herramienta de trabajo Google Colab (Google Colaboratory)

Una **GPU** es una **Unidad de Procesamiento Gráfico**, que es principalmente usada por los gamers o los desarrolladores de videojuegos. Una GPU es más poderosa que una CPU, pues su arquitectura está pensada para procesar de manera eficiente datos más complejos.

En 2013 Google comenzó a desarrollar la primera versión de lo que hoy se conoce como **TPU (Tensor Processing Unit**, o unidad de procesamiento de tensores). Esta es una arquitectura de computador pensada específicamente para el desarrollo de modelos de Machine Learning

Herramienta de trabajo Google Colab (Google Colaboratory)

Google Colab permite ejecutar código Python en la nube. Para ello usa el formato de Jupyter Notebook, que es simplemente como una versión de Python enriquecida: además del código Python, es posible agregar texto con diferentes formatos o imágenes.

Para utilizar Google Colab es necesario tener una cuenta de Google. En nuestro caso crearemos la cuenta en Google:

nombre.apellido.aepi@gmail.com

en el siguiente enlace:

<https://accounts.google.com/signup/v2/createaccount?continue=https%3A%2F%2Faccounts.google.com%2FManageAccount%3Fnc%3D1&biz=false&theme=glif&flowName=GlifWebSignIn&flowEntry=SignUp>



Crea una cuenta de Google

Introduce tu nombre

Nombre

Pedro

Apellidos (opcional)

Bonillo Ramos

Siguiente

Español (España)



[Ayuda](#)

[Privacidad](#)

[Términos](#)



Información básica

Introduce tu fecha de nacimiento y tu sexo.

Día	Mes	Año
<input type="text" value="2"/>	<input type="text" value="Septiemb"/>	<input type="text" value="1972"/>
Género		
<input type="text" value="Hombre"/>		

Siguiente

Español (España)



Ayuda

Privacidad

Términos



Elige tu dirección de Gmail

Elige una dirección de Gmail o crea otra

☐ pbonilloramos@gmail.com

☐ bonilloramosp@gmail.com

☒ Crear dirección de Gmail personalizada

Crea una dirección de Gmail

pedro.bonillo.aepi

@gmail.com

Puedes utilizar letras, números y puntos

[Usar tu correo electrónico](#)

[Siguiente](#)



Crea una clave segura

Crea una contraseña segura con una combinación de letras, números y símbolos

Contraseña

.....

Confirmación

.....

☐ Mostrar contraseña

Siguiente

Español (España)



Ayuda

Privacidad

Términos



Añade un correo de recuperación

Dirección en la que Google puede ponerse en contacto contigo si detecta actividad inusual en tu cuenta o no puedes iniciar sesión.

Dirección de correo electrónico de recuperaci...
pbonillo@gmail.com

Siguiente

Saltar

Español (España)



Ayuda

Privacidad

Términos



Añadir el teléfono



Número de teléfono

615702581

Google solo usará este número para mantener la seguridad de la cuenta. No lo mostrará a otros usuarios. Más tarde podrás elegir si quieres que se use con otros fines.

Siguiente

Saltar

Español (España)

Ayuda

Privacidad

Términos



Verifica tu teléfono

Para comprobar que el número sea tuyo, te enviaremos un SMS con un código de verificación de 6 dígitos. *SMS sujeto a tarifas estándar*

 615 70 25 81

[Atrás](#)

[Ahora no](#)

[Enviar](#)

Español (España)



[Ayuda](#)

[Privacidad](#)

[Términos](#)



Verifica tu teléfono

Para comprobar que este número es tuyo, Google te enviará un SMS con un código de verificación de 6 dígitos.

 615 70 25 81

Introduce el código de verificación

G- 322460

Se aplicarán tarifas estándar.

[Atrás](#)

[Verificar](#)

Español (España)



[Ayuda](#)

[Privacidad](#)

[Términos](#)



Verifica tu teléfono

Para comprobar que este número es tuyo, Google te enviará un SMS con un código de verificación de 6 dígitos.

 615 70 25 81

Introduce el código de verificación

G- 322460

Se aplicarán tarifas estándar.

[Atrás](#)

[Verificar](#)

Español (España) ▼

[Ayuda](#)

[Privacidad](#)

[Términos](#)



Sácale el máximo partido a tu número

Si quieres, puedes añadir tu número de teléfono a tu cuenta para usarlo en los servicios de Google. [Más información](#)

Por ejemplo, tu número se utilizará para

☐ Recibir videollamadas y mensajes

☒ Hacer que los servicios de Google, incluidos los anuncios, te resulten más relevantes


[Más opciones](#)

[Atrás](#)



Revisa la información de tu cuenta

Puedes usar esta dirección de correo o este número de móvil para iniciar sesión en otro momento

 Pedro Bonillo Ramos
pedro.bonillo.aepi@gmail.com

Número de móvil de recuperación
615 70 25 81

Siguiente

Español (España)



[Ayuda](#)

[Privacidad](#)

[Términos](#)



Elige los ajustes de personalización



Personalización rápida (1 paso)

Usa ajustes de personalización que ofrecen contenido y anuncios personalizados. Te recordaremos que revises tus ajustes en un par de semanas.



Personalización manual (5 pasos)

Configura los ajustes de personalización paso a paso. Tú decides qué ajustes quieres activar y cuáles no para tener la experiencia de contenido y anuncios que prefieras.

Puedes cambiar la configuración en cualquier momento en account.google.com

Siguiente

funcion de tus ajustes, en Google y en la Web

Con respecto al contenido y los anuncios no personalizados, lo que veas puede estar influido por factores como lo que estés viendo en ese momento y tu ubicación (el servicio de anuncios se basa en la ubicación general). El contenido y los anuncios personalizados se pueden basar, además de en esos criterios, en tu actividad, como las búsquedas que hagas en Google y los vídeos que veas en YouTube. Los anuncios y el contenido personalizados incluyen, entre otras cosas, resultados y recomendaciones más relevantes, una página de inicio de YouTube personalizada y anuncios adaptados a tus intereses.

Puedes cambiar la configuración del navegador para rechazar algunas o todas las cookies.



Recordatorio de privacidad

Te enviaremos un recordatorio en un par de semanas para que revises estos ajustes

Atrás

Confirmar

Español (España)



Ayuda

Privacidad

Términos

derechos y su seguridad o se dañen sus bienes en la medida exigida o permitida por la ley, lo que incluye la divulgación de información a autoridades gubernamentales.

- Llevar a cabo investigaciones que mejoren nuestros servicios para los usuarios y beneficien al público en general.
- Cumplir las obligaciones que hemos asumido frente a nuestros partners como, por ejemplo, desarrolladores y titulares de derechos.
- Hacer valer nuestros derechos legales, lo que incluye la investigación de posibles infracciones de los Términos del Servicio aplicables.

Puedes ir a tu cuenta de Google (account.google.com) para hacer una Revisión de Privacidad o ajustar los controles de privacidad.

¿Tienes alguna duda? [Contacta con nosotros](#)

[Cancelar](#)







[Acepto](#)

Español (España) ▼


[Ayuda](#)

[Privacidad](#)

[Términos](#)

 Inicio Información personal Datos y privacidad Seguridad Contactos y compartir Pagos y suscripciones Información general[Privacidad](#) [Términos](#) [Ayuda](#)[Información](#)

Bienvenido, Pedro Bonillo Ramos

Gestiona tu información, la privacidad y la seguridad para mejorar tu experiencia en Google. [Más información](#) 

Privacidad y personalización

Consulta los datos almacenados en tu cuenta de Google y elige qué actividad se debe guardar para personalizar tu experiencia en Google

[Gestionar tus datos y tu privacidad](#)

Hay consejos de seguridad para ti

Consejos de seguridad encontrados en la Revisión de Seguridad

[Revisar consejos de seguridad](#)

<https://colab.research.google.com/?hl=es>

colab.research.google.com/?hl=es

Te damos la bienvenida a Colaboratory

Archivo Editar Ver Insertar Entorno de ejecución Herramientas Ayuda

Índice

Primeros pasos

Ciencia de datos

Aprendizaje automático



Más recursos

Ejemplos destacados

Sección

Abrir cuaderno

Buscar cuadernos

Título	Abierto por última vez ▲	Abierto por primera vez ▼	
 <u>Te damos la bienvenida a Colaboratory</u>	6:41	6:41	

+ Nuevo cuaderno

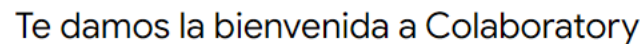
Cancelar

Conectar ▼

^

Compartir

P

 $\{x\}$

-

< >

25

Por ejemplo, a continuación se muestra una **celda de código** con una breve secuencia de comandos de Python que calcula un valor, lo almacena en una variable e imprime el resultado:

Herramienta de trabajo Google Colab (Google Colaboratory)

- Los cuadernos de Colab permiten combinar **código ejecutable y texto enriquecido** en un mismo documento, además de **imágenes, HTML, LaTeX** y mucho más.
- Los cuadernos que se crean en Colab se almacenan en la cuenta de Google Drive. Se puede compartir los cuadernos de Colab fácilmente con compañeros de trabajo o amigos, lo que permite comentarlos o incluso editarlos.
- Para crear un cuaderno de Colab, se debe usar el menú Archivo que aparece arriba o bien acceder al enlace para [crear un cuaderno de Colab](#).
- Los cuadernos de Colab son cuadernos de Jupyter alojados en Colab.



Nuevo cuaderno

Ctrl+O

Subir cuaderno

Guardar una copia en Drive

Guardar una copia como Gist de GitHub

Guardar una copia en GitHub

Guardar

Ctrl+S

Descargar

Imprimir

Ctrl+P

```
[ ] import numpy as np
    from matplotlib import pyplot as plt

    ys = 200 + np.random.randn(100)
```

o **ejecutable** y **texto enriquecido** en un mismo documento, además de **imágenes**, **HTML**,
Colab se almacenan en tu cuenta de Google Drive. Puedes compartir tus cuadernos de Colab
que les permite comentarlos o incluso editarlos. Consulta más información en [Información](#)
b, puedes usar el menú Archivo que aparece arriba o bien acceder al enlace para [crear un](#)
alojados en Colab. Para obtener más información sobre el proyecto Jupyter, visita

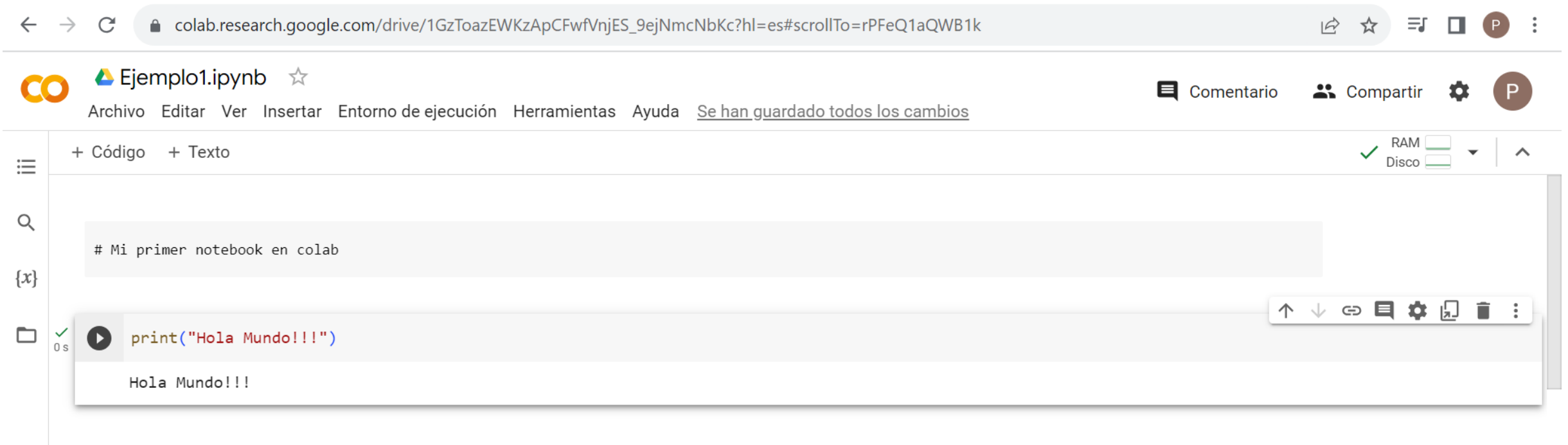
Imprimir Ctrl+P

Las bibliotecas más populares de Python para analizar y visualizar datos. La celda de código de abajo utiliza **Numpy** para generar datos aleatorios y **Matplotlib** para visualizarlos. Para editar el código, solo tienes que hacer clic en la celda.

✓ 0 s completado a las 7:20





Herramienta de trabajo Google Colab (Google Colaboratory)



The screenshot displays the Google Colab web interface. At the top, the browser address bar shows the URL `colab.research.google.com/drive/1GzToazEWKzApCFwfVnjES_9ejNmcNbKc?hl=es#scrollTo=rPFQ1aQWB1k`. Below the browser, the Colab header includes the logo, the notebook name "Ejemplo1.ipynb", and a star icon. A menu bar contains "Archivo", "Editar", "Ver", "Insertar", "Entorno de ejecución", "Herramientas", and "Ayuda". On the right, there are links for "Comentario", "Compartir", and a settings gear. A status bar at the top right indicates "Se han guardado todos los cambios".

The main workspace shows a notebook with two cells. The first cell is a text cell containing the comment `# Mi primer notebook en colab`. The second cell is a code cell containing the Python code `print("Hola Mundo!!!")`. The code cell has a play button icon on the left and a toolbar on the right with icons for undo, redo, link, comment, settings, insert, and delete. Below the code cell, the output "Hola Mundo!!!" is displayed. On the left sidebar, there are icons for a menu, search, a variable `{x}`, and a file explorer showing a folder with a green checkmark and "0 s".

RAM 
Disco 

Fundamentos generales de Python para la ciencia de datos

¿Por qué Python?

- **Legibilidad:**
 - Sintaxis limpia
 - Fácil de comprender
- **Rápida codificación:**
 - Estructuras de datos amigables.
 - Lenguaje listo para ejecutar (**interpretado**)
 - Interprete de python interactivo
- **Reusabilidad:**
 - Permite compartir funcionalidad entre programas usando módulos y paquetes
- **Portabilidad:**
 - Python tiene la misma interfaz en múltiples plataformas: Linux, Windows, MacOS, etc.
- **Código Abierto, Gratis, y Desarrollo Comunitario:**
 - La especificación y la implementación es open-source.
- **Orientado a Objeto:**
 - Conceptos y características disponibles pero no obligatorio.

Fundamentos generales de Python para la ciencia de datos

- **Tipado dinámico:**
 - Una misma variable puede tomar valores de distinto tipo en distintos momentos.
- **Resolución dinámica de nombres:**
 - Métodos, variables son enlazadas con su lógica durante la ejecución.
- **Biblioteca estándar de Python es extensa y bien documentada:**
 - Conocido como "batteries included"
- **Gran variedad de librerías de terceros:**
 - Más de 130.000 paquetes en un gran rango de funcionalidades, incluyendo: GUI, Web, Multimedia, Base de datos, Redes, Testing, Automatización, Web Scraping, Procesamiento de texto, Procesamiento de imágenes, etc.
- **Sponsorizado por grandes empresas:**
 - Facebook, Google, Amazon, Redhat, Microsoft, etc.

Legibilidad

- Más uso de palabras:
 - `!`, `||`, `&&` → not, or, y and
- **Indentación (sangrado):**
 - En lugar de usar llaves o palabras claves para delimitar bloques de código, usa espacios o tabuladores.

Bash

```
if [ $x -ge 18 ]; then
    echo "Es mayor"
else
    echo "Es menor"
fi
```

Python

```
if x >= 18: print("Es
            mayor")
else:
    print("Es      menor")
```

Comentarios y Variables

- **Comentarios:**

```
'''
```

```
Comentario más largo en una línea en Python '''
```

```
print("Hola mundo") # Al final de una línea de código
```

- **Variables:**

- Las variables no necesitan ser declaradas, no se tiene que especificar cuál es su tipo.
- Las variables son creadas cuando se les asigna valores. Se usa el símbolo = para asignar valores.
- Deben ser asignadas antes de ser usadas.
- Se les puede asignar cualquier tipo de dato.

```
x = 1
```

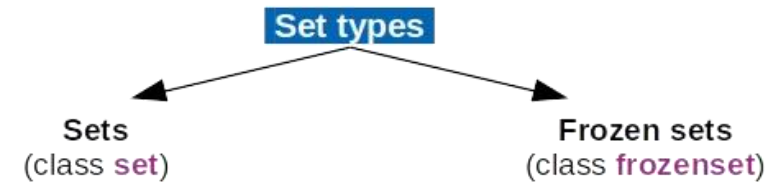
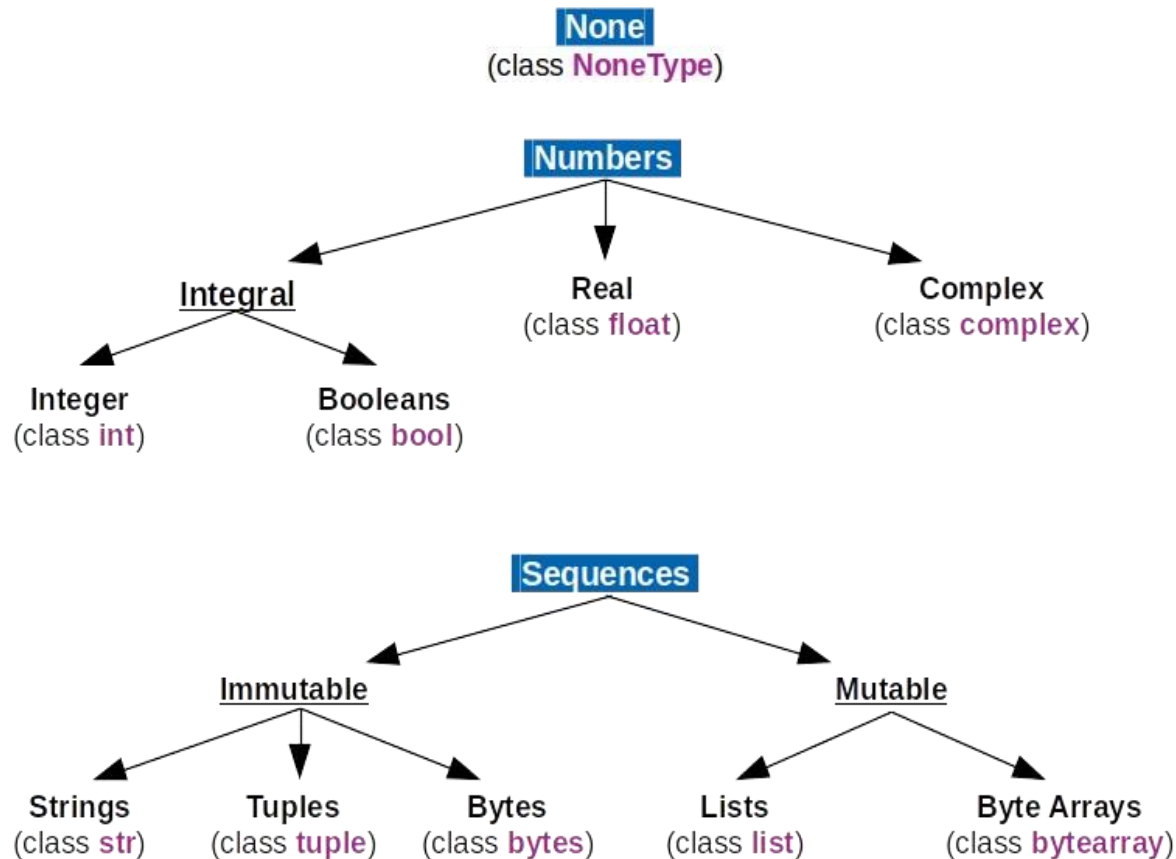
```
x = "texto" # Posible porque los tipos son  
# asignados dinámicamente
```

Tipo de datos

Tipo	Clase	Notas	Ejemplo
NoneType	None	Representa la ausencia de valor	None
bool	Numbers	Valor booleano verdadero o falso	True o False
int	Numbers	Número entero de tamaño ilimitado	42
float	Numbers	Número real; coma flotante	3.1415927
complex	Numbers	Número complejo con parte real y parte imaginaria j	4.5 + 3j
str	Sequences	Cadena en formato unicode. Inmutable	"Texto"
list	Sequences	Secuencia de datos, pueden ser de diversos tipos. Mutable.	[4.0, 'Cadena', True]
tuple	Sequences	Secuencia de datos, pueden ser de diversos tipos. Inmutable.	(4.0, 'Cadena', True)
set	Set Types	Conjunto de datos, sin orden, no contiene duplicados. Mutable	set([4.0, 'Cadena', True])
frozenset	Set Types	Conjunto de datos, sin orden, no contiene duplicados. Inmutable	frozenset([4.0, 'Cadena', True])
dict	Mappings	Diccionario de pares clave:valor (array asociativo)	{'key1': 1.0, 'key2': False}
bytearray	Binary Sequences	Secuencia de bytes. Mutable	bytearray([119, 105, 107, 105])
bytes	Binary Sequences	Secuencia de bytes. Inmutable	bytes([119, 105, 107, 105])

Jerarquía de Tipos de Datos

Python 3
The standard type hierarchy



Mappings

Dictionaries
(class dict)

Callable

< Functions, Methods, Classes >

Modules

Tipos Numéricos

- Números enteros (int):
 - Decimal: 24, 60
 - Binario: 0b010011, 0b1101
 - Hexadecimal: 0x18, 0x3cf4
 - Octal: 0o30, 0o74
- Números de punto flotante (float): números reales y la precisión depende del equipo.
 - 3.141595
 - 12.
 - -45.3556
 - 2,0/3,0
- Números complejos (complex):
 - $6.32 + 45j$
 - $0.117j$
 - $(2 + 0j)$
 - $1j$
- Valores booleanos (bool): Se usa para expresiones lógicas
 - False (equivale al número 0)
 - True (cualquier otro valor diferente de cero y 1 por defecto)

String y None

- **str** (cadena de caracteres):
 - 'Wikipedia'
 - "Wikipedia"
 - """Con
múltiples
líneas"""
- **None**:
 - El tipo None representa un valor "vacío".
 - a = None

Listas

- **Listas (array indexado):**

- Es la secuencia más general en python.
- Mutables; se puede cambiar su contenido en tiempo de ejecución.
- Para declarar una lista se usan los corchetes [] y los elementos se separan por comas.
- Pueden contener elementos de diferentes tipos.
- No tienen un tamaño fijo.
- Los elementos son ordenados por la posición.
- Para acceder a los elementos se utiliza un índice entero (empezando por "0", no por "1"). Se pueden utilizar índices negativos para acceder elementos a partir del final.

Listas

- Crear una lista:

```
lista = ["abc", 42, 3.1415]
```

- Acceder a un elemento por su índice:

```
lista[0]
```

```
'abc'
```

- Acceder a un elemento usando un índice negativo:

```
lista[-1]
```

```
3.1415
```

- Añadir un elemento al final de la lista:

```
lista.append(True)
```

```
lista
```

```
['abc', 42, 3.1415, True]
```

Listas

- Re-asignar el valor del primer elemento de la lista:

```
lista[0] = "xyz"
```

- Borrar un elemento de la lista:

```
lista.remove(True)
```

```
del lista[0]
```

- Mostrar una sublista:

```
lista[0:2]    # Del índice "0" al "2" (sin incluir este último)
```

```
['xyz', 42]
```

- Listas anidadas (una dentro de otra):

```
lista_anidada = [lista, [True, 42]]
```

```
lista_anidada
```

```
[['xyz', 42, 3.1415], [True, 42]]
```

```
lista_anidada[1][0]
```

```
True
```

Listas

```
lista = [22, True, "a list", [1, 2]]
```

```
lista[0] =>??
```

```
lista[2][4] =>??
```

```
lista[-1][-2] =>??
```

```
lista[0:3] =>??
```

```
lista[:3] =>??
```

```
lista.append('DevOps') =>??
```

```
lista.insert(0, "a list") =>??
```

```
lista.remove("a list") =>??
```

Listas

```
lista = [22, True, "a list", [1, 2]]
```

```
lista[0] => 22
```

```
lista[2][4] => 's'
```

```
lista[-1][-2] => 1
```

```
lista[0:3] => [22, True, 'a list']
```

```
lista[:3] => [22, True, 'a list']
```

```
lista.append('DevOps') => [22, True, 'a list', [1, 2], 'DevOps']
```

```
lista.insert(0, "a list") => ['a list', 22, True, 'a list', [1, 2], 'DevOps']
```

```
lista.remove("a list") => [22, True, 'a list', [1, 2], 'DevOps']
```

Tuplas

- **Tuplas:**

- Es otra secuencia en python como las listas.
- **Inmutables**; no se puede cambiar su contenido en tiempo de ejecución.
- Para declarar una lista se usan los **paréntesis ()** y los elementos se separan por comas. **Es necesario que tengan como mínimo una coma. También se pueden declarar sin los paréntesis.**
- Pueden contener elementos de diferentes tipos.
- Pueden definirse de cualquier tamaño.
- Los elementos son ordenados por la posición.
- Para acceder a los elementos se utiliza un índice entero (empezando por "0", no por "1"). Se pueden utilizar índices negativos para acceder elementos a partir del final.

Tuplas

- Crear una tupla:

```
tupla = ("abc", 42, 3.1415)
```

- Acceder a un elemento por su índice:

- `tupla[0]`

`'abc'`

- Acceder a un elemento usando un índice negativo:

- `tupla[-1]`

`3.1415`

- No es posible modificar la tupla:

- `del tupla[0]`

(Excepción)

- `tupla[0] = "xyz"`

(Excepción)

Tuplas

- Mostrar una sub-tupla:

```
tupla[0:2]  # Del índice "0" al "2" (sin incluir este último)
('abc', 42)
```

- Tuplas anidadas (una dentro de otra):

```
tupla_anidada = (tupla, (True, 3.1415))
(('abc', 42, 3.1415), (True, 3.1415))
tupla_anidada[1][0]
True
```

- También es una tupla:

```
1, 2, 3, "abc"
(1,) #Ojo (1) no es una tupla
(1, 2,)
```


Tuplas

- La inmutabilidad se puede omitir si una nueva estructura es enlazada a la tupla original
 - `>>> t = 10,15,20`
 - `>>> t = t[0],t[2]`
 - `>>> t`
 - `(10,20)`

Diccionarios

- **Diccionarios (array asociativo):**

- **Mutables**; se puede cambiar su contenido en tiempo de ejecución.
- Para declarar un diccionario se usan las llaves `{}`. Contienen elementos separados por comas, donde cada elemento está formado por un par **clave:valor** (el símbolo `:` separa la clave de su valor correspondiente).
- Las claves de un diccionario deben ser inmutables (strings, números, o tuplas)
- El valor asociado a una clave puede ser de cualquier tipo de dato, incluso un diccionario.
- No tienen un tamaño fijo.
- Indexados por la clave.

Diccionarios

- Crear un diccionario:

```
diccionario = {"cadena": "abc", "numero": 42, "lista": [True, 42]}
```

- Acceder a un elemento por su clave:

- `diccionario["cadena"]`

`'abc'`

- `diccionario["lista"][0]`

`True`

- Insertar un nuevo elemento clave:valor:

```
diccionario["decimal"] = 3.1415927
```

Diccionarios

- Re-asignar el valor del primer elemento de la lista:

```
diccionario["cadena"] = "xyz"
```

- Borrar un elemento de la lista:

- del diccionario["cadena"]

- También es posible que un valor sea un diccionario

```
diccionario_mixto = {"tupla": (True, 3.1415), "diccionario": diccionario}
```

```
diccionario_mixto["diccionario"]["lista"][1]
```

Diccionarios

```
dic = {'e': 2.718, 'pi': 3.141, 'fi': 1.618}
```

```
>>> dic['e']
```

```
?
```

```
# Actualizar el valor de pi a 3.141592
```

```
>>> ?
```

```
>>> dic.keys()
```

```
?
```

```
>>> dic.values()
```

```
?
```

```
>>> dic.items()
```

```
?
```

Diccionarios

```
dic = {'e': 2.718, 'pi': 3.141, 'fi': 1.618}
>>> dic['e']
2.718

# Actualizar el valor de pi a 3.141592
>>> dic['pi'] = 3.141592
>>> dic.keys()
dict_keys(['e', 'pi', 'fi'])
>>> dic.values()
dict_values([2.718, 3.141, 1.618])
>>> dic.items()
dict_items([('e', 2.718), ('pi', 3.141), ('fi', 1.618)])
```

Diccionarios

- Operaciones comunes con Diccionarios:
 - Agregar un elemento (update):
 - `dic.update({"d":4})`
 - Crear una copia del diccionario (copy):
 - `nuevodic = dic.copy()`
 - Eliminar todos los elementos de un diccionario:
 - `dic.clear()`

Conjuntos

- **Conjuntos:**

- Los conjuntos se construyen mediante **set(items) / frozenset(items)** donde items es cualquier objeto iterable, como listas o tuplas.
 - set para conjuntos mutables y
 - frozenset para conjuntos inmutables.
- Los conjuntos no mantienen el orden ni contienen elementos duplicados.
- Se suelen utilizar para eliminar duplicados de una secuencia, o para operaciones matemáticas como intersección, unión, diferencia y diferencia simétrica.

Conjuntos

- Crear conjuntos:

```
conjunto1 = set(["a", "b", "a"])
```

```
conjunto2 = set(["a", "b", "c", "d"])
```

```
conjunto_inmutable = frozenset(["a", "b", "a"])
```

- Intersección

```
conjunto1 & conjunto2
```

```
set(['a', 'b'])
```

- Unión

```
conjunto1 | conjunto2
```

```
set(['a', 'c', 'b', 'd'])
```

Conjuntos

- Diferencia (1)

conjunto1 - conjunto2

set([])

- Diferencia (2)

conjunto2 - conjunto1

set(['c', 'd'])

- Diferencia simétrica

conjunto1 ^ conjunto2

set(['c', 'd'])

Bytes, Bytearray

- Para manejar datos binarios python incluye los tipos bytes y bytearray.
- El tipo bytes es una secuencia inmutable de bytes, conceptualmente similar a una cadena. Y el tipo bytearray es una secuencia mutable de bytes.
- Representan a un carácter conforme a su número correspondiente en el código ASCII y se definen anteponiendo la letra b a los apostrofes o comillas.

b'<texto>'

b"<texto>"

Bytes

- Ejemplos:

```
palabra = b"Hola" // palabra = bytes([72,111,108,97])
```

```
palabra[0]
```

```
72
```

```
palabra[2:4]
```

```
b'la'
```

- str -> bytes.

```
bytes('hola', "utf-8")
```

- bytes -> str

```
str(b'hola'[1:3], 'ascii')
```

Bytearray

- Crear un bytearray desde un bytes:

```
x = bytearray(b"Python Bytes")
```

- Crear un bytearray desde un string:

- `x = bytearray("Python Bytes", "utf8")`

- Crear un bytearray desde una lista de enteros:

- `x = bytearray([94, 91, 101, 125, 111, 35, 120])`

Rebanadas de secuencias (slice)

- `secuencia[x:y:z]`
 - Desde x, hasta y sin incluir dicha posición y con incrementos de z
- Ejemplos de `t=(1,2,3,4,5)`
 - `t[2:4]` → Desde 2 hasta 3
 - `t[:3]` → Desde el comienzo hasta 2 `t[3:]` → Desde 3 hasta el final
 - `t[:]` → Desde el comienzo hasta el final `t[::-1]` → En sentido inverso
 - `t[::-2]` → En sentido inverso con incrementos de 2

Condicional: if

- If
 - ∴ if sintaxis:
 - Las clausulas elif y else son opcionales.
 - No existe el operador case en python; se puede usar la estructura if/elif/else.

```
if <condition>:  
    <statements>  
[elif <condition>:  
    <statements>]  
[elif <condition>:  
    • pass]  
...  
[else:  
    <statements>]
```

Condicional: if

- Ejemplos:

- `a=7`

- `if a>6:`

- `print("Es mayor que 6")`

- `if 1: # 1 significa verdadero`

- `print("sip")`

- `if a == 1:`

- `print("1")`

- `elif a == 2:`

- `print("2")`

- `else:`

- `print("Mayor a 2")`

Condicional: if

- **if anidados:**

```
a=1
```

```
b=2
```

```
if a==1:
```

```
    if b==2:
```

```
        print("a es 1 y b es 2")
```

- **if en una línea**

```
if a>4: print("Greater")
```

Bucle: for

```
for <variable> in <secuencia>:
```

```
    <sentencias>
```

```
[else:
```

```
    <sentencias>]
```

- La sentencia else es opcional y siempre se ejecuta al menos que se ejecute la sentencia break dentro del bucle.
- Se itera por cada valor en la secuencia, en cada iteración la variable tomando el valor correspondiente en la secuencia.
- Las secuencias pueden ser: una lista, una tupla, un diccionario, un conjunto o un string.

Bucle: for

- Ejemplos:
- #Iterando con una lista:

```
frutas = ["banana", "uva"]  
for x in frutas:  
    print(x)
```
- #Iterando con cadenas:

```
for x in "banana":  
    print(x)
```
- #usando la función range (genera una secuencia de números):

```
for i in range(6):  
    print(i, end=', ')
```

Bucle: while

while <condición>:

<sentencias>

[**else**:

<sentencias>]

- La sentencia else es opcional y siempre se ejecuta al menos que se ejecute la sentencia break dentro del bucle.
- Se itera mientras la condición sea verdadera.
- Es importante modificar dentro del bucle los elementos que forman la condición para finalizar las iteraciones.

Bucle: while

- Ejemplos:

- `i = 1`

```
while i < 6:
```

```
    print(i)
```

```
    i += 1
```

- #En una línea

```
a=3
```

```
while a>0: print(a); a-=1
```

- `a=3`

```
while(a>0):
```

```
    print(a)
```

```
    a-=1
```

```
else:
```

```
    print("a llego a 0")
```

Bucle anidados

- #Bucles **for** anidados:

```
for i in range(1,6):  
    for j in range(i):  
        print("*",end=' ' )  
    print()
```

- #Bucles **while** anidados:

```
i=6  
while(i>0):  
    j=6  
    while(j>i):  
        print("*",end=' ' )  
        j-=1  
    i-=1  
    print()
```

Control de bucles

- Modificar el comportamiento normal de los bucles en python: continue y break.
- **continue:**
 - detiene la iteración actual y continua con la siguiente.
- **break:**
 - detiene la iteración actual y todas las restantes.
- En python no se puede especificar que bucle anidado se pretende controlar.

Control de bucles

- #break

```
for i in 'break':  
    print(i)  
    if i=='a': break;
```

- # continue

```
i=0  
while(i<8):  
    i+=1  
    if(i==6): continue  
    print(i)
```


Control de bucles

#Con bucles anidados

```
for x in range(10):  
    for y in range(10):  
        print (x*y)  
        if x*y > 50:  
            break  
    else:  
        continue  
break
```

Funciones

```
def <NombreDeLafunción>(arg1, arg2, ...):  
    """<Texto>""" #Docstring: muy recomendado  
    <sentencias>  
    return <data>  
  
<NombreDeLafunción>(arg1, arg2, ...)
```

- Las funciones ayuda a dividir un programa en módulos. Hace al código más fácil de administrar, depurar y escalar. Reutilización del código.
- Se puede acceder el texto de documentación utilizando el atributo `__doc__` de la función.
- `return` nos permite retornar un valor.

Funciones

- `def hola():`

`"""`

`Esto es el docstring de hola`

`"""`

`print("Hola")`

`hola.__doc__`

`hola()`

- `#Con parámetros:`

`def suma(a,b):`

`print(a+b)`

`suma(1,2)`

- `# Con un valor de retorno`

`def func1(a):`

`if a%2==0:`

`return 0`

`else:`

`return 1`

`func1(7)`

Funciones

- Parámetros (argumentos):
 - No es necesario especificar el tipo de objeto de un argumento,
 - Los argumentos tienen un comportamiento posicional, pero ...
 - ... python también habilita pasar argumentos usando el nombre y su valor independientemente del orden.
 - Se puede especificar valores de los parámetros por defecto. Entonces si no se pasa un argumento, se usa el valor por defecto.
 - Los objetos mutables se pasan por referencia.
 - Los objetos inmutables se pasan por valor.
 - Los módulos, clases, instancias y otras funciones se pueden usar como argumentos y son examinados dinámicamente.
 - La cantidad de argumentos puede ser indefinido.

Funciones

- #Con valores por defecto:
- #Mutables x referencia:

```
def suma(a=1,b=3):
```

```
    print(a+b)
```

```
suma(1,2)
```

```
suma()
```

```
suma(a=7)
```

```
suma(b=10,a=5)
```

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

```
def fun(a):
```

```
    a[0]=3
```

```
fun(lista)
```

```
lista[0]
```

. Funciones

- `def suma(a=1,b=3):`
 `print(a+b)`
- `def sumar(f):`
 `f(2,2)`
- # Función como argumento.
`sumar(suma)`

#Arg indeterminados - Posición

```
def ind_posicion(*names):  
    for name in names:  
        print("Hola " + name)  
ind_posicion("Carlos","Marta")
```

#Arg indeterminados –Nombre

```
def ind_nombre(**kwargs):  
    print (kwargs)  
ind_nombre(n=5, c="Hola", l=[1,2,3])
```

Funciones

- Retorno:
 - Detiene la ejecución de una función.
 - Puede ser una expresión.
 - Se pueden devolver múltiples valores usando tuplas.
 - Cuando una función no tiene ninguna sentencia de retorno, se devuelve implícitamente el valor “None”.

```
def sum(a,b):  
    return a+b
```

```
def test():  
    return 'abc', 100
```

Funciones

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

- Enlaces a más información:
 - [Liberia estándar](#)
 - [Doc en español](#)

Operadores Aritméticos

Operador	Descripción
+	Suma
-	Resta
-	Negativo
*	Multiplicación
**	Exponente
/	División
//	División entera
%	Residuo

Operadores de Asignación

Operador	Descripción	Ejemplo
=	Asignación simple	$x = y$
+=	Suma	$x += y$ equivale a $x = x + y$
-=	Resta	$x -= y$ equivale a $x = x - y$
*=	Multiplicación	$x *= y$ equivale a $x = x * y$
**=	Exponente	$x ** = y$ equivale a $x = x ** y$
/=	División	$x /= y$ equivale a $x = x / y$
//=	División entera	$x //= y$ equivale a $x = x // y$
%=	Residuo de división	$x \% = y$ equivale a $x = x \% y$

Operadores de Relación

Operador	Evalúa
<code>==</code>	$a == b$ ¿a igual a b?
<code>!=</code>	$a != b$ ¿a distinta de b?
<code>></code>	$a > b$ ¿a mayor que b?
<code><</code>	$a < b$ ¿a menor que b?
<code>>=</code>	$a >= b$ ¿a mayor o igual que b?
<code><=</code>	$a <= b$ ¿a menor o igual que b?

Operadores Lógicos.

Operador	Descripción	Ejemplo
and	¿se cumple a y b?	<pre>>>> True and False False</pre>
or	¿se cumple a o b?	<pre>>>> True or False True</pre>
not	No al valor	<pre>>>> not True False</pre>

Operadores de Identidad

Operador	Evalúa
<i>is</i>	<i>a is b</i> Equivale a $id(a) == id(b)$
<i>is not</i>	<i>a is not b</i> Equivale a $id(a) != id(b)$

Operadores de Pertenencia

- **In**: evalúan si un objeto se encuentra dentro de otro
"ni" in "Daniel"
True
- **not in** evalúan si un objeto **no** se encuentra dentro de otro.
"ni" not in "Daniel"
False

Operadores para objetos de tipo str

Operador	Descripción
+	Concatenación
*	Repetición

Operadores de bits

Operador	Descripción
&	AND
	OR
^	XOR
<<	Mover x bits a la izquierda
>>	Mover x bits a la derecha

Operadores de bits

Operador	Descripción
&	AND
	OR
^	XOR
<<	Mover x bits a la izquierda
>>	Mover x bits a la derecha

Identificadores

- "Un identificador es un nombre definido por el usuario para representar una variable, una función, una clase, un módulo o cualquier otro objeto". 5 reglas para nombrar identificadores en Python:
 - 1) Puede ser una combinación de letras minúsculas (a-z) / mayúsculas (A- Z), dígitos (0-9) o un guión bajo (_).
 - 2) No puede empezar con un dígito.
 - 3) No podemos usar símbolos especiales (! @ # \$ % .)
 - 4) No se puede usar las palabras reservadas.
 - 5) Puede tener cualquier largo.
- Nota: Python diferencia entre mayúsculas y minúsculas.

Identificadores

- Palabras reservadas:

```
>>> import keyword
```

```
>>> keyword.kwlist
```

and	def	False	import	not	True
as	del	finally	in	or	try
assert	elif	for	is	pass	while
break	else	from	lambda	print	with
class	except	global	None	raise	yield
continue	exec	if	nonlocal	return	

Manejo de variables

- Asignación simple (=)

```
>>> a=5
```

- Asignación multiple:

```
>>> edad,nombre = 21, 'Daniel'
```

- Mismo valor para múltiples variables:

- ```
>>> age=fav=7
```

- Intercambiando variables

```
>>> a,b='red','blue'
```

```
>>> a,b=b,a
```

- Borrando variables

```
>>> del a
```

# Conversión de tipos

- Para convertir entre tipos de datos se puede utilizar:
  - `bool(x)` Convierte x en un booleano.
  - `int (x)` Convierte x en un entero
  - `float (x)` Convierte x en un número real
  - `str (x)` Convierte x a una cadena.
  - `set(x)` Convierte x en un conjunto
  - `list(x)` Convierte x en una lista
  - `tuple(x)` Convierte x en una tupla
- Nota: hay conversiones que no se pueden realizar y retorna un error.

# Variables Locales y Globales

- Variables Locales:
  - Una variable creada dentro de una función pertenece al ámbito local de esa función y solo se puede usar dentro de esa función.
- Variables Globales:
  - Una variable creada en el cuerpo principal del código es una variable global y pertenece al ámbito global.
  - Se puede usar la palabra clave "global" cuando desee tratar una variable como global en un ámbito local.

# Variables Locales y Globales

- #con errores

```
a=0
```

```
def func():
```

```
 print(a)
```

```
 a=1
```

```
 print(a)
```

```
func()
```

- #ámbitos

```
def red():
```

```
 a=1
```

```
 def blue():
```

```
 b=2
```

```
 print(a)
```

```
 print(b)
```

```
 blue()
```

```
 print(a)
```

```
red()
```

- #global

```
a=1
```

```
def counter():
```

```
 global a
```

```
 a=2
```

```
 print(a)
```

```
counter()
```

```
a
```

- #nonlocal

```
def red():
```

```
 a=1
```

```
 def blue():
```

```
 nonlocal a
```

```
 a=2
```

```
 b=2
```

```
 print(a)
```

```
 print(b)
```

```
 blue()
```

```
 print(a)
```

```
red()
```

# Formateadores de strings

- Para imprimir variables junto con una cadena se puede usar comas o usar formateadores de cadenas.
- **Comas:**  

```
nombre="Carlos"
edad=45

print(nombre, "tiene", edad, "años de edad")
```
- **f-strings (nuevo formado, recomendado):**
  - ```
print(f"{nombre} tiene {edad} años de edad")
```
 - La letra "f" precede a la cadena, y las variables se mencionan entre llaves en sus lugares.

Formateadores de cadenas

- **Método format():**

- `print("{0} tiene {1} años de edad".format(nombre,edad))`
- `print("{a} tiene {b} años de edad".format(a=nombre,b=edad))`
- Sucede a la cadena y las variables van como argumentos separados por comas. Se usa llaves para colocar las variables. Se puede hacer referencia a la posición o al nombre si se definen.

Formateadores de cadenas

- **Operador %:**

- `print("%s tiene %s años de edad"%(nombre,edad))`
- Se colocan donde van las variables en una cadena. %s es para la cadena. Lo que sigue a la cadena es el operador % y las variables en una tupla.
- Otras opciones incluyen:
 - %d - para enteros
 - %f - para números de coma flotante

Entrada estándar

- La función `input()` permite obtener texto escrito por teclado. El programa se detiene esperando que se escriba algo y se pulse la tecla Intro.
 - `print("¿Cómo se llama? ", end="")`
`nombre = input()`
`print(f"Me alegro de conocerle, {nombre}")`
 - `nombre = input("¿Cómo se llama? ")`
`print(f"Me alegro de conocerle, {nombre}")`
- De forma predeterminada, la función `input()` convierte la entrada en una cadena, aunque escribamos un número. Si intentamos hacer operaciones, se producirá un error.

Entrada estándar

- Conversión de tipos

- `cantidad = input("Dígame una cantidad en euros:")`
`print(f"{cantidad} euros son {round(cantidad * 1.15, 2)} dólares")`
- `cantidad = int(input("Dígame una cantidad en euros:"))`
`print(f"{cantidad} euros son {round(cantidad * 1.15, 2)} dólares")`

Manejo de archivos

- Abrir / Crear un archivo:
 - Lo primero es abrir (o crear) un archivo.
 - `open(nombre, modo)`
 - Nombre: es el nombre del archivo.
 - Modos:
 - `r` para lectura (por defecto)
 - `w` para escritura
 - `r+` o `w+` lectura escritura
 - `a` para agregar al final
 - `T` modo texto (por defecto)
 - `b` modo binario.

Manejo de archivos

- Crear el archivo:

Crear una archivo en modo write

```
f = open("hola.txt", "w")
```

Escribir un texto al archivo

```
f.write("Hola Mundo!")
```

Cerrar el arhivo

```
f.close()
```

Manejo de archivos

- Leer un archivo:

Abrir el archivo en modo 'read'

```
f = open("hola.txt", "r")
```

Colocar el contenido del archivo en una variable

```
f_contenido= f.read()
```

Cerrar el archivo

```
f.close()
```

Imprimir el contenido del archivo

```
print(f_contenido)
```

Manejo de archivos

- Agregar a un archivo:

```
f = open("hola.txt", "a")
```

```
f_contenido = "\r\n" + "Agregando una linea!"
```

```
f.write(f_contenido)
```

```
f.close()
```


Manejo de archivos

- Leer una línea **del** archivo

```
f = open("hola.txt", "r")
```

```
linea1 = f.readline()
```

```
linea2 = f.readline()
```

```
print("Línea 1:", linea1)
```

```
print("Línea 2:", linea2)
```

```
f.close()
```

Manejo de archivos

- Leer un archivo línea a línea:

```
f = open("hola.txt", "r")
```

```
for linea in f:
```

```
    print("Linea:", linea, end="")
```

```
f.close()
```

```
#Using 'list()'
```

```
f = open("hola.txt", "r")
```

```
f_content = list(f)
```

```
print(f_content)
```

```
f.close()
```

Manejo de archivos

- Sentencia **with**:

```
with open("hola.txt", "w") as f:  
    # Escribir un texto al archivo  
    f.write("Hola Mundo!")
```

- Nota: con **with** el close se hace automáticamente

Manejo de archivos

Method	Description
<u>close()</u>	Closes the file
<u>detach()</u>	Returns the separated raw stream from the buffer
<u>fileno()</u>	Returns a number that represents the stream, from the operating system's perspective
<u>flush()</u>	Flushes the internal buffer
<u>isatty()</u>	Returns whether the file stream is interactive or not
<u>read()</u>	Returns the file content
<u>readable()</u>	Returns whether the file stream can be read or not
<u>readline()</u>	Returns one line from the file
<u>readlines()</u>	Returns a list of lines from the file
<u>seek()</u>	Change the file position
<u>seekable()</u>	Returns whether the file allows us to change the file position
<u>tell()</u>	Returns the current file position
<u>truncate()</u>	Resizes the file to a specified size
<u>writable()</u>	Returns whether the file can be written to or not
<u>write()</u>	Writes the specified string to the file
<u>writelines()</u>	Writes a list of strings to the file

Programación Orientada a Objeto (POO)

- Los estilos de programación se les llama paradigmas:
 - Secuencial o lineal:
 - Las instrucciones van de arriba hacia abajo, no tenemos que abstraer cosas complejas, simplemente damos ordenes una tras otra.
 - Para aplicaciones sencillas suele ser muy directo y mantenible.
 - Estructurado:
 - Surge con la idea de mejorar la claridad, calidad y tiempo de desarrollo de una programación secuencial. Se basa en en subrutinas y estructuras básicas:
 - bloques de código
 - sentencias condicionales
 - y bucles.

Programación Orientada a Objeto (POO)

- Lineal y Estructurado:
 - Se centran en parte algorítmica y la lógica de programación más que en la representación de los datos y la descripción de la lógica del negocio. Los datos están separados y sin relación con los procedimientos o funciones.
 - Aunque es útil para muchos problemas para el desarrollo de grandes aplicaciones con una lógica de negocio extensa y dinámica es complejo de mantener y evolucionar.

Programación Orientada a Objeto (POO)

- POO:
 - Es la evolución natural de la programación estructurada. Se basa en una representación más cercana a como expresaríamos las cosas en la vida real.
 - Se basa en dividir el programa en pequeñas unidades lógicas de código. A estas **pequeñas unidades lógicas de código se les llama objetos**.
 - Los objetos son estructuras independientes que **combinan datos** o estados **(variables) con** acciones asociadas o **comportamiento (métodos)**.
 - La lógica de negocio se modela e implementa mediante una serie de objetos que **interactúan entre si**.
 - Este enfoque aumenta la capacidad para administrar la complejidad del software, lo cual resulta especialmente importante cuando se desarrollan y mantienen aplicaciones y estructuras de datos de gran tamaño.

Programación Orientada a Objeto (POO)

- Características principales:
 - **Abstracción:** Los objetos permiten modelar las características esenciales y el comportamiento de entidades reales sin revelar "cómo" se implementan.
 - **Encapsulamiento:** Consiste en agrupar en una clase las características y el comportamiento de un mismo nivel a abstracción.
 - **Ocultamiento:** Es la capacidad de ocultar los detalles internos de una clase y exponer sólo los detalles que sean necesarios para el resto del sistema.
 - **Polimorfismo:** Se refiere a la propiedad de invocar acciones sintácticamente iguales a objetos de tipos distintos.

Programación Orientada a Objeto (POO)

- Características principales:

- **Herencia:**

- Las clases no se encuentran aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.

- **Recolección de basura:**

- Es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente los objetos que hayan quedado sin ninguna referencia a ellos.
 - Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno asignará memoria al crear un nuevo objeto y la liberará la memoria cuando nadie esté usando el objeto.

POO en Python

- Características principales:
 - **Abstracción:** soportado
 - **Encapsulamiento:** soportado
 - **Ocultamiento:** brinda esta característica por convención.
 - **Polimorfismo:** natural por su enfoque dinámico (enlazado tardío).
 - **Herencia:** soporta herencia simple y múltiple
 - **Recolección de basura:** soportado

POO: Clases

- Una clase es un tipo de dato definido por el usuario. La clase es como el plano (la definición) de los objetos. Sintaxis:

- **# Clase base:**

```
class <NombreDeLaClase>:
```

```
    """documentación"""
```

```
    <Sentencias de la clase>
```

- **# Herencia (simple y multiple)**

```
class <NombreDeLaClase>(<ClaseBase1, ..., ClaseBaseN>):
```

```
    """documentación"""
```

```
    <Sentencias de la clase>
```

POO: Clases

- Crear una clase:

```
class MiClase:  
    x = 5
```

- Crear un objeto:

```
o = MiClase()  
print(o.x)
```

- Borrar un objeto:
del o

- Todo es un objeto:

```
[1]: a=5  
    type(a)
```

```
[1]: int
```

```
[4]: a=int(5)  
    type(a)
```

```
[4]: int
```

POO: Clases: `__init__()`

- Todas las clases tienen una función llamada `__init__()`, que siempre se ejecuta cuando se instancia la clase.
- Se le llama “constructor” y se usa para asignar valores a los atributos del objeto u otras operaciones que sean necesarias cuando se crea el objeto.
- Nota: El parámetro `self` es una referencia a la instancia actual de la clase y se usa para acceder a las variables que pertenecen a la clase.

```
class Persona:
```

```
    def __init__(self, nombre, edad):  
        self.nombre = nombre  
        self.edad = edad
```

```
p1 = Persona("Carlos", 36)
```

```
print(p1.nombre)
```

```
print(p1.edad)
```

POO: Clases: Métodos

- Los objetos pueden contener métodos (son funciones que pertenecen al objeto).
- Todos los métodos de clase deben tener el argumento adicional `self` como primer argumento.

```
class Persona:  
    def __init__(self, nombre, edad):  
        self.nombre = nombre  
        self.edad = edad  
  
    def mimetodo(self):  
        print("Hola, mi nombre es " + self.nombre)
```

```
p1 = Persona("Carlos", 36)  
p1.mimetodo()
```

POO: Clases: Variables

- Las variables del objeto se pueden modificar o borrar de la siguiente manera:

```
class Persona:
    def __init__(self, edad):
        self.edad = edad

p1 = Persona(36)
# Modificar
p1.edad = 40
# Borrar
del p1.edad
```

POO: Clases: Variables

- Dentro de una clase hay dos tipos principales de variables:
 - Variables de la clase:
 - Tienen el mismo valor en todas las instancias de la clase (es decir, variables estáticas)
 - Se accede a la variable de la clase usando:
<NombreDeLaClase>.<NombreDeLaVariable>
 - Variables de la instancia:
 - Suelen tener valores diferentes en todas las instancias de la clase.
 - Se accede a la variable de la instancia usando
<NombreDeLaInstancia>.<NombreDeLaVariable>

POO: Clases: Variables

```
class Auto:
```

```
    # Variable de la clase
```

```
    ruedas = 4
```

```
    def __init__(self, marca):
```

```
        # Variable de la instancia
```

```
        self.marca = marca
```

```
auto1= Auto("Renault")
```

```
auto2= Auto("Suzuki")
```

```
print(f"Marca: {auto1.marca}, Ruedas: {Auto.ruedas}")
```

```
print(f"Marca: {auto2.marca}, Ruedas: {Auto.ruedas}")
```

POO: Clases: Variables

- Cuando se crean variables de instancia realmente se generan dos, una de la clase y otra de instancia con el mismo nombre. Y cual se está usando depende de como se acceda.

```
class Auto:  
    ruedas = 4  
    def __init__(self, marca):  
        self.marca = marca
```

```
instancia= Auto("Renault")  
instancia.ruedas = 5  
print(instancia.ruedas)  
print(Auto.ruedas)
```

POO: Encapsulamiento

- Nada en Python es privado. Todos los atributos de Python (variables y métodos) son públicos.
- Por convención se usa un solo guión bajo antes del nombre para señalar que es un atributo interno y no debería usarse externamente.

```
class Auto:
```

```
    def __init__(self, marca):
```

```
        self._marca = marca
```

```
auto = Auto("Renault")
```

```
auto._marca
```

POO: Encapsulamiento

- Un doble guión bajo__al principio de una variable lo “hace privado”. Da una fuerte sugerencia de no tocarlo desde fuera de la clase. Cualquier intento de acceder dará como resultado un AttributeError ya que python cambia el nombre al siguiente formato:

<_NombreDeLaClase>__<NombreDeLaVariable>.

```
class Auto:
    def __init__(self, marca):
        self.__marca = marca
auto= Auto("Renault")
auto._Auto__marca
```

POO: Herencia

- La herencia nos permite definir una clase que hereda todos los métodos y propiedades de otra clase.
 - La clase padre es la clase de la que se hereda, también llamada clase base.
 - La clase hija es la clase que hereda de otra clase, también llamada clase derivada

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
    def printNombre(self):
        print(self.nombre)
class Estudiante(Persona):
    pass
x = Estudiante("Alejandro")
x.printNombre()
```

POO: Herencia

```
class Persona:
```

```
    def __init__(self, nombre):
```

```
        self.nombre = nombre
```

```
class Estudiante(Persona):
```

```
    def __init__(self, nombre, fecha): # Constructor propio
```

```
        super().__init__(nombre) # Para mantener la herencia
```

```
        self.fechagrduacion = fecha
```

```
    def welcome(self):
```

```
        print(f"Bienvenido {self.nombre} ({self.fechagrduacion})")
```

```
x = Estudiante("Alejandro", "2019")
```

```
x.welcome()
```

POO: Herencia Múltiple

```
class A:  
    def a(self):  
        print('b')
```

```
class B:  
    def b(self):  
        print('c')
```

```
class C(A, B):  
    def c(self):  
        print('d')
```

```
c = C()
```

```
c.a()
```

```
c.b()
```

```
c.c()
```

Manejo de Excepciones

- Hay dos tipos de errores durante la ejecución de un programa:
 - Errores de sintaxis:
 - Los errores de sintaxis ocurren cuando se escribe el código incorrectamente.
 - En tales casos, la línea errónea es repetida por el analizador con una flecha apuntando a la primera ubicación en donde el error fue detectado.
 - Excepciones:
 - Estos ocurren durante la ejecución de un programa cuando algo inesperado sucede. Por ejemplo, división por cero.
 - Cuando no estás manejando excepciones apropiadamente, el programa se cerrará de manera abrupta ya que no sabe que hacer en tales casos.

Manejo de Excepciones

try:

<sentencias>

except excepción1 [as variable1]:

<sentencias>

...

except excepciónN [as variableN]:

<sentencias>

except (exA, exB, ...) [as variable]:

<sentencias>

except:

<sentencias>

else:

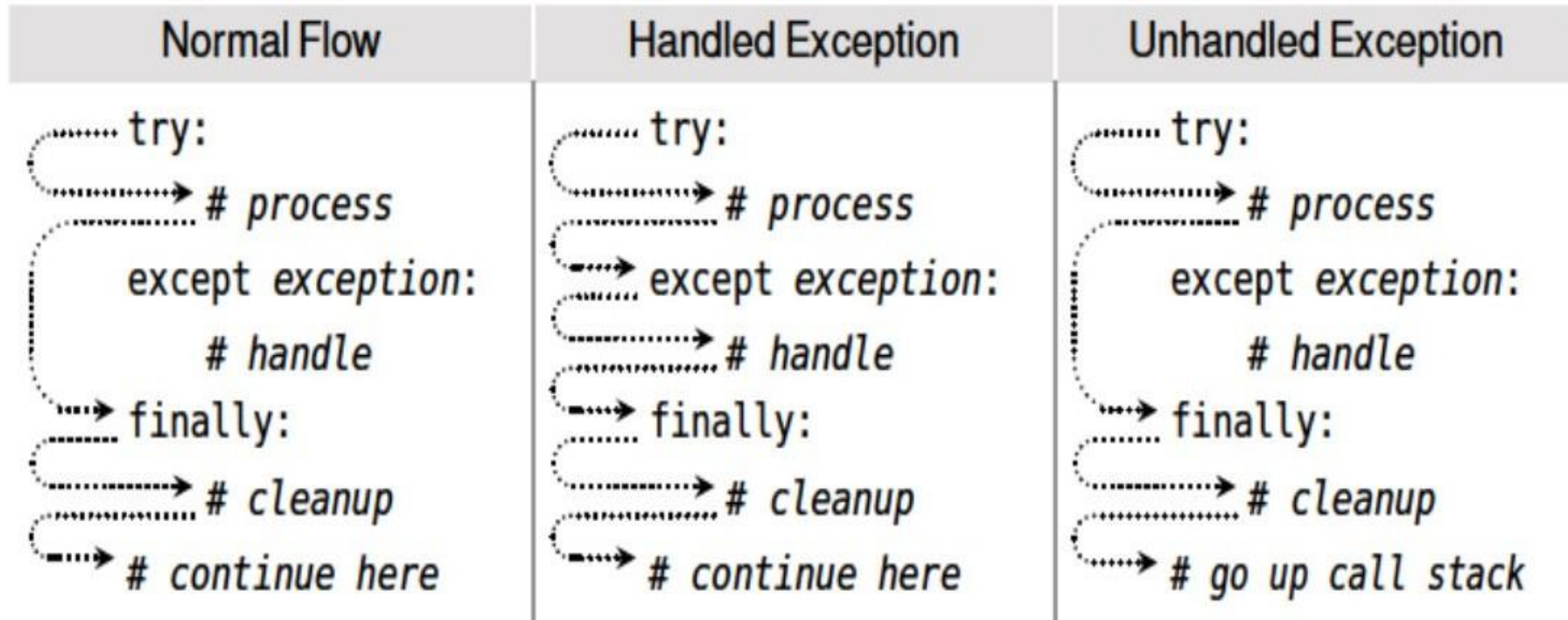
<sentencias>

finally:

<sentencias>

- **Try:** permite controlar las excepciones dentro de un bloque de código.
- **Except:** permite ejecutar código si ocurrió alguna excepción.
- **Else:** permite ejecutar código si no ocurrieron excepciones.
- **Finally:** permite ejecutar código independientemente de si ocurrieron o no excepciones.

Manejo de Excepciones



Manejo de Excepciones

```
while True:
```

```
    try:
```

```
        x = int(input("Ingrese un número: "))
```

```
    except ValueError:
```

```
        print("El valor ingresad no es un entero.")
```

```
    else:
```

```
        print("Calculando 50 /", x, "Resultado:", 50/x)
```

```
    finally:
```

```
        print("Ya hice todo lo necesario.")
```

Manejo de Excepciones

- En ciertas ocasiones es deseable generar una excepción:
 - Si estamos dentro de un bloque except, se puede lanzar una excepción sin especificar que excepción.
raise
 - Especificando un excepción (estándar o personalizada) y un argumento:
raise <TipoDeError>(<mensaje>)

```
x = -1
```

```
if x < 0:
```

```
    raise Exception("x tiene un valor negativo")
```

Manejo de Excepciones

- Las excepciones personalizadas son clases que heredan de otra excepción:

```
class exceptionName(baseException):  
    <sentencias>
```

- Ejemplo:

```
class MiException(Exception):  
    def __init__(self, mensaje):  
        super().__init__(mensaje)
```

```
raise MiException("Usando una excepción personalizada")
```