# Introducción a python

Luis, Gutierrez cinema.nightmare@gmail.com
Carlos Rodríguez carlosrdz.isd@gmail.com

March 7, 2020

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Constantes y variables

# Constantes y variables

- Una constante es un valor que se mantiene fijo durante todo el programa. Ejemplos: 5, 'a', 38, "www.google.com"
- Una variable es un espacio de almacenamiento en memoria que contiene cierta información, la cuál puede ser modificada durante la ejecución del programa. Ejemplos: variable, a, a5 = 32, var = "hola"

# Tipos de datos

# Tipos de datos

- Entero (integer)
- Flotante (float)
- Número complejo
- Booleano (boolean)
- String
- Lista
- Tupla
- Diccionario

# Operadores aritméticos

# Operadores aritméticos

- + (suma)
- - (resta)
- * (multiplicación)
- / (división)
- % (módulo)
- ** (exponente)
- // función suelo

# Operadores lógicos

# Operadores lógicos

- >(mayor)
- <(menor)
- == (igual a)
- != (diferente de)
- >= (mayor o igual)
- <= (menor o igual)

# Operadores lógicos

- and (y)
- or (o)
- not
- y otros...

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Hola python!

# Hola python!

- # En este programa desplegamos un saludo
- print(”Hola grupo de python!”)

# Comentarios

# Comentarios

- Los comentarios sirven para hacer más entendible el código, tanto para nosotros como para los demás
- Python sabe que es un comentario al escribir el hashtag al inicio de la línea
- # Así por ejemplo
- Otra forma de escribir un comentario es con triple comilla doble al inicio y al final del comentario
- """Este es otro ejemplo de
- comentarios"""

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Especificando a python

# Especificando a python

- Para que python sepa qué tipo de dato deseamos utilizar, es necesario realizar un ”cast” al tipo de dato. En caso de no especificar el tipo de dato que deseamos, python otorgará el que el crea más apropiado a la variable, por ejemplo:
- str(texto)
- int(numero)
- float(texto)

# Especificando a python

- Para que python sepa qué tipo de dato deseamos utilizar, es necesario realizar un ”cast” al tipo de dato. En caso de no especificar el tipo de dato que deseamos, python otorgará el que el crea más apropiado a la variable, por ejemplo:
- str(texto)
- int(numero)
- float(texto) #esto es válido ;)

# Números

# Números

- Los dos tipos principales de números en python son los enteros (integers) y reales (floats)
- En caso de no especificar el tipo de número (o de dato) python decide cuál es el más apropiado (a veces se equivoca)
- Integers (int): 2, 5, 4000, -2
- Floats (float): 2.5, 2.9, -3.1

# Texto

# Texto

- Una variable que almacena texto se llama cadena o string, y para que python las reconozca deben estar en comillas simples o dobles... o triples si necesita varias líneas Ejemplo: ”Texto de ejemplo”
- el operador + une dos strings...

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# input y print

# input y print

- Para desplegar algún dato de nuestro programa, basta con utilizar el comando print, mientras que para asignar algún valor dado por el usuario a una variable, se utiliza el comando input. Ejemplos:
- print(”Desplegando un texto”)
- var=input(”Ingresa un texto”)
- print(var)

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Arreglos (colecciones)

# Arreglos (colecciones)

- Existen 4 tipos de arreglos en python (legacy), los cuáles permiten almacenar varios datos en una sola variable:
- Listas (List): Ordenada e intercambiable. Permite miembros duplicados.
- Tuplas (Tuple): Coleccion ordenada y no intercambiable. Permite miembros duplicados.
- Conjuntos (Set): Desordenada y no indexada. No permite miembros duplicados.
- Diccionarios (Dictionary): Desordenada, intercambiable e indexada. No permite miembros duplicados.

# Listas

# Listas

- La lista es una colección ordenada e intercambiable.
- Para crear una lista en python se escriben sus elementos entre corchetes [a, b, c,”hola”].
- mi_lista=[”un_elemento”,”otro_elemento”,”tercer_elemento”, ”otro”, ”otro”, ”ya”, ”son”, ”muchos”]
- print(mi_lista)
- Para accesar a una lista, especificamos el número de indexación al que queremos accesar (el primer elemento se indentifica con el número 0).
- print(mi_lista[1])

# Listas

# Listas

- python permite indexamiento negativo...

# Listas

- python permite indexamiento negativo... no lo usen )=
- Es posible accesar a una sublista de un rango especificado
- print(mi_lista[2:5])
- print(:5)

# Operaciones de listas

# Operaciones de listas

- Las operaciones más habituales de Python son las siguientes:
- lista[i]: Devuelve el elemento que está en la posición i de la lista.
- lista.pop(i): Devuelve el elemento en la posición i de una lista y luego lo borra.
- lista.append(elemento): Añade elemento al final de la lista.
- lista.insert(i, elemento): Inserta elemento en la posición i.
- lista.extend(lista2): Fusiona lista con lista2.
- lista.remove(elemento): Elimina la primera vez que aparece elemento.

# Diccionarios

# Diccionarios

- Un diccionario (tal como los convencionales) es una palabra que tiene asociado ”algo”, y a diferencia de las listas, los diccionarios no tienen orden.
- Para crear un diccionario se ponen sus elementos entre llaves {”a”:”Almidon”,”b”:”bueno”,:}. se les denomina llaves (keys) a las palabras (a y b) y valores a sus definiciones (Almidon y bueno). No es posible tener dos llaves iguales, aunque si dos valores.
- Ejemplo:
- diccionario = {’Piloto 1’:’Fernando Alonso’, ’Piloto 2’:’Kimi Raikkonen’, ’Piloto 3’:’Felipe Massa’}
- print(diccionario)

# Operaciones más comunes de los diccionarios

# Operaciones más comunes de los diccionarios

- diccionario.get('key'): Devuelve el valor que corresponde con la key introducida.
- diccionario.pop('key'): Devuelve el valor que corresponde con la key introducida, y luego borra la key y el valor.
- diccionario.update({'key':'valor'}): Inserta una determinada key o actualiza su valor si ya existiera.
- "key" in diccionario: Devuelve verdadero (True) o falso (False) si la key (no los valores) existe en el diccionario.
- "definicion" in diccionario.values(): Devuelve verdadero (True) o falso (False) si definición existe en el diccionario (no como key).

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Conjuntos

# Conjuntos

- También existen los conjuntos (sets) aunque son menos utilizados.
- Tal como los diccionarios, se crean usando llaves, pero sus elementos se separan por coma como si se tratara de una lista
- Ejemplo: conjunto = {'Fernando Alonso', 'Kimi Raikkonen', 'Felipe Massa'}
- Los sets permiten realizar operacionas matemáticas típicas de conjuntos como unión, intersección, etc.

# Operaciones más comunes en conjuntos

# Operaciones más comunes en conjuntos

- A — B: Unión entre el conjunto A y B (Los elementos del conjunta A y los elementos del conjunto B)
- A & B: Intersección entre el conjunto A y B (los elementos que están en ambos conjuntos)
- A – B: Diferencia entre el conjunto A y B (los elementos que están en A pero no están en B)
- A^B: Diferencia simétrica entre el conjunto A y B (los elementos que están en A o en B pero no en los dos)

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Indentación

# Indentación

- Python considera es sensible a la indentación, por lo que es necesario que sus líneas de código se encuentren agrupadas con el mismo número de espacios a la izquierda de cada línea. Es recomendado utilizar bloques de cuatro espacios, sin embargo cualquier otra cantidad de espacios (o tabuladores) pueden ser utilizados de igual forma (no se recomienda).

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Control de flujo y condicionales

# Control de flujo y condicionales

- Todo programa con una ligera complejidad necesita ”tomar decisiones” en algunas bifurcaciones del problema, donde, dependiendo de alguna condición se realiza una u otra acción.
- Esta bifurcación se lleva a cabo con el comando if (condición principal), con sus opcionales elif (condiciones adicionales, pueden ponerse tantos como sean deseados) y else (el default en caso de no cumplirse ninguno).

# Ejemplo

# Ejemplo

- Revisar ejercicio ”ifs.py”

# Condiciones en python

# Condiciones en python

- Las condiciones utilizadas con más frecuencia son:
- a == b –>Indica si a es igual a b
- a <b
- a >b
- not –>NO: niega la condición que le sigue.
- and –>Y: junta dos condiciones que tienen que cumplirse las dos
- or –>O: junta dos condiciones y tienen que cumplirse alguna de las dos.

# while en python

# while en python

- Cuando una tarea debe repetirse hasta que cierta condición se cumpla (no sabemos cuántas veces se repetirá), es recomendado utilizar el comando while, que se usa de la siguiente forma:
- vuelta=1
- while vuelta <10:
- print(”vuelta ”+str(vuelta))
- vuelta=vuelta+1

# for en python

# for en python

- En ocasiones necesitamos repetir varias veces una determinada tarea, pero conocemos la cantidad de veces que la repetiremos. Para esto es recomendado utilizar en Python el comando for, que se usa de la siguiente forma:
- for vuelta in range(1,10):
- print(”vuelta ”+str(vuelta))
- PD. En el caso del for (a diferencia del while) no es posible realizar un loop infinito.
- es posible utilizar el for con cualquier objeto que pueda ser iterado (recorrer sus elementos)

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# No reinventes la rueda

# No reinventes la rueda

- Un módulo es un programa que viene en un ”paquete” de python y existen para contener rutinas que son utilizadas habitualmente y que no sea necesario reescribirlas cada que son necesitadas. Estos módulos son creados muchas veces por programadores como tu, y se comparten en Github principalmente asi que si hay alguna rutina que uses mucho y no existe ningun paquete que la contenga, es tu oportunidad de brillar ;)
- Gracias a estos módulos es posible realizar cosas complejas en pocas líneas de código

# Instalando un módulo

# Instalando un módulo

- Los módulos son instalados de forma distinta en los diversos sistemas operativos, en el caso de Linux, basta con bajar el programa pip y escribir en nuestra línea de comandos ”pip install <módulo>” para instalarlo... en algunas distribuciones es incluso más fácil que eso, pero recuerda que internet es tu amigo (internet, google no)

# Módulos ”legacy” más usados

# Módulos ”legacy” más usados

- os
- datetime
- time
- sys
- locale
- MySQLdb
- Cada uno de estos módulos contiene funciones ya incluídas que pueden ser llamadas si esl módulo se encuentra instalado y cargado.

# Invocando funciones de módulos

# Invocando funciones de módulos

- Para llamar a una función de un módulo es necesario primero cargar el módulo a nuestro programa, con el comando:
- import modulo
- Posteriormente basta con llamar a la función deseada en la forma:
- modulo.funcion()

# Declaración de funciones

# Declaración de funciones

- Cuando existe alguna rutina que queremos repetir constantemente durante nuestro programa, es útil crear una función, la cuál nos permitirá llamar esta rutina con un solo comando (en lugar de escribir todo). Ejemplo:
- def myfunc(): #Aquí especificamos el nombre de la función
- x = 300
- print(x)
- myfunc() #Esto llama a la función

# Scope

# Scope

- Una variable se encuentra disponible solamente dentro de la región donde fue creada, y a esto de le denomina ”scope” o ”alcance”.
- Una variable que es creada dentro de una función pertenece solamente al ”scope local” de esa función, y solamente podrá ser usada dentro de esa función
- ***Recuerda que la indentación afecta al scope de las variables

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Interactuando con archivos

# Interactuando con archivos

- Con Python podemos manipular archivos (leer, escribir...) de una forma bastante sencilla

# Creando archivos de texto

# Creando archivos de texto

- Para crear un archivo hay que seguir los siguientes pasos:
- Paso 1) Abre un archivo y asignalo a una variable (con la opción "w+")
- Paso 2) Escribimos la informacion que queramos que contenga el archivo
- Paso 3) Cerramos el archivo

# Creando archivos de texto (Ejemplo)

# Creando archivos de texto (Ejemplo)

- Paso 1) f=open(”archivo.txt”, ”w+”)
- Paso 2) for i in range(10):
- f.write(”This is line %d \r \n” % (i+1))
- Paso 3) f.close()

# Escribir datos sobre un archivo

# Escribir datos sobre un archivo

- Para escribir sobre un archivo hay que seguir los siguientes pasos:
- Paso 1) Abre un archivo y asignalo a una variable (con la opción ”a+”)
- Paso 2) Escribe lo que desees con el comando <variable >.write(”texto”)
- Paso 3) Cerramos el archivo

# Escribir datos sobre un archivo (Ejemplo)

# Escribir datos sobre un archivo (Ejemplo)

- Paso 1) f=open(”archivo.txt”, ”a+”)
- Paso 2) for i in range(2):
- f.write(”Appended line %d \r \n” % (i+1))
- Paso 3) f.close()

# Leer un archivo

# Leer un archivo

- Para Leer un archivo hay que seguir los siguientes pasos:
- Paso 1) Abre un archivo y asignalo a una variable (con la opción "r+")
- Paso 2) Leemos la información deseada con el comando <variable >.read()... y es probable que quieras almacenarla en alguna variable
- Paso 3) Cerramos el archivo

# Leer un archivo (Ejemplo)

# Leer un archivo (Ejemplo)

- Paso 1) f=open(”archivo.txt”, ”r”)
- Paso x) if f.mode == ’r’: #esto es opcional, pero recomendado
- Paso 2) contents =f.read()
- Paso 3) f.close()

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Clases y métodos

# Clases y métodos

- Python es un lenguaje ”orientado a objetos”. Esto implica que casi todo el código es implementado utilizando constructores especiales llamados clases. Los programadores utilizan clases para mantener las cosas relacionadas juntas. Esto se ase con la palabra reservada ”class”, que es un grupo de constructores orientado a objetos.

# ¿Qué es una clase?

# ¿Qué es una clase?

- Una clase es una plantilla para crear objetos. Los objetos tienen variables miembros y tienen un comportamiento asociado a ellas. En python una clase es creada por la palabra reservada ”class”.
- Un objeto es creado utilizando el constructor de la clase. Este objeto será llamado entonces como una instancia de la clase. En python creamos instancias de la siguiente forma:

```
Instance = class(arguments)
```

# ¿Cómo crear una clase?

# ¿Cómo crear una clase?

- La clase más simple puede ser creada con la palabra reservada class. Por ejemplo, podemos crear una clase simple y vacía sin ninguna funcionalidad.

```
>>> class Snake:
...     pass
...
>>> snake = Snake()
>>> print(snake)
<__main__.Snake object at 0x7f315c573550>
```

# Atributos y métodos

# Atributos y métodos

- Una clase por si misma no tiene ningún uso (como la anterior) a menos que tenga alguna funcionalidad asociada a ella. Las funcionalidades son definidas asignando atributos, que actúan como contenedores de información y funciones relacionadas a esos atributos. Esas funciones son llamadas métodos.

# Atributos

# Atributos

- Puedes definir la siguiente clase con el nombre Ejemplo. Esta clase tendrá el atributo ”name”.

```
>>> class Ejemplo:
...     name = ”python” # asigna el atributo ‘name‘ de la clase
...
```

- Puedes asignar la clase a una variable. Esto es llamado instanciación de un objeto. Con esto podrás accesar a los atributos presentes dentro de la clase utilizando el operador punto. Por ejemplo, en el ejemplo Ejemplo, puedes accesar al atributo name de la clase Ejemplo.

```
>>> # instanciar la clase Ejemplo y asignarla a la variable ejemplo
>>> ejemplo = Ejemplo()

>>> # accesar al atributo de la clase name dentro de la clase Ejemeplo
>>> print(ejemplo.name)
python
```

# Métodos

# Métodos

- Una vez que haya atributos que ”pertenezcan” a la clase, puedes definit las funcciones que acceden a los atributos de la clase. Esas funciones son llamadas métodos. Cuando defines métodos, necesitas proveer el priimer argumento del métodos con la palabra reservada self.
- Por ejemplo, puedes definir la clase Ejemplo, que tiene un atributo name y un método change_name. El método change_name tomará el argumento new_name junto con la palabra reservada self.

```
>>> class Ejemplo:
...     name = "python"
...
...     def change_name(self, new_name): # nota que el primer argumento es self
...         self.name = new_name # accesa al atributo de  la clase con la palabra
...
```

# Métodos

# Métodos

- Ahora, puedes iniciar esta clase Ejemplo con una variable ejemplo y cambiar el nombre con el método change_name

```
>>> # instancia la clase
>>> ejemplo = Ejemplo()

>>> # imprime el name actual del objeto
>>> print(ejemplo.name)
python

>>> # cambia name utilizando el metodo change_name
>>> ejemplo.change_name("anaconda")
>>> print(ejemplo.name)
anaconda
```

# Instanciando atributos y el método init

# Instanciando atributos y el método init

- También es posible proveer los valores de los atributos en tiempo de ejecución. Esto se hace definiendo los atributos dentro del método init. El siguiente ejemplo ilustra esto.

```
class Ejemplo:

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

    def change_name(self, new_name):
        self.name = new_name
```

# Instanciando atributos y el método init

# Instanciando atributos y el método init

- Ahora puedes definir directamente los valores de los atributos para objetos separados. Por ejemplo:

```
>>> # dos variables son instanciadas
>>> python = Snake("python")
>>> anaconda = Snake("anaconda")

>>> # imprime los names de las dos variables
>>> print(python.name)
python
>>> print(anaconda.name)
anaconda
```

# Herencia en python

# Herencia en python

- La herencia es el proceso en que una clase toma atributos y métodos de otra. La nueva clase formada es llamada clase hijo, y las clases hijo son derivados de una clase padre.
- Es importante considerar que las clases hijos pueden sobreescribir o extender la funcionalidad (atributos y comportamientos) de la clase padre. En otras palabras, la clase hijo hereda todos los atributos y comportamientos del padre pero también puede tener distinto comportamiento

# Instanciando atributos y el método init

# Instanciando atributos y el método init

- Cuando defines una nueva clase en Python 3, implícitamente se utiliza object como la clase padre. Por lo que las dos definiciones siguientes son equivalentes:

```
class Dog(object):
    pass

# En Python 3, esto equivale a:

class Dog:
    pass
```

# Herencia en python

# Herencia en python

- La herencia es el proceso en que una clase toma atributos y métodos de otra. La nueva clase formada es llamada clase hijo, y las clases hijo son derivados de una clase padre.
- Es importante considerar que las clases hijos pueden sobreescribir o extender la funcionalidad (atributos y comportamientos) de la clase padre. En otras palabras, la clase hijo hereda todos los atributos y comportamientos del padre pero también puede tener distinto comportamiento

# Extendiendo la funcionalidad de la clase padre

# Extendiendo la funcionalidad de la clase padre

- Ver dog_inheritance.py

# Sobreescribiendo la funcionalidad de una clase padre

# Sobreescribiendo la funcionalidad de una clase padre

- Ver dog_inheritance2.py

# Encapsulación

# Encapsulación

- Python sigue la filosofía de que todos aquí somos adultos con respeto de ocultar atributos y métodos, permitiendo ocultarlos de otros programadores cuando hagan uso de tus clases (utiliza atributos planos siempre que sea posible).
- Un miembro protejido en C++ y java es accesible solo dentro de la clase y sus subclases. Y para lograr esto suficiente agregar el nombre de estas con doble guionbajo __, y cone sto estás diciendo a los demás "no toques esto a menos que seas una subclase.

# Encapsulación (ejemplo)

```
class Person:
  def __init__(self):
    self.name = 'Manjula'
    self.__lastname = 'Dube'

  def PrintName(self):
    return self.name +' ' + self.__lastname

#Outside class
P = Person()
print(P.name)
print(P.PrintName())
print(P.__lastname)
#AttributeError: 'Person' object has no attribute '__lastname'
```

# Encapsulación (otro ejemplo)

```
class BicicletaMontana(Bicicleta):
    """Esta es una bicicleta de montana"""
    micolor = ''
    def __init__(self, color):
        self.micolor = color
        Bicicleta.__init__(self, color=color, tipo="Montana")
bicicleta_mtn = BicicletaMontana('rojo')
bicicleta_mtn.micolor
>>> 'rojo'
bicicleta_mtn.__color
>>> AttributeError: 'BicicletaMontana' object has no attribute '__color'
```

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Utilizando strings

# Utilizando strings

- Como sabemos, es posible desplegar un string con la función print directamente, y asignarle valores como a cualquier otra variable, es decir:

```
a = "hola"
print(a)
```

# Utilizando strings

# Utilizando strings

- Como sabemos, es posible desplegar un string con la función print directamente, y asignarle valores como a cualquier otra variable, es decir:

```
a = "hola"
print(a)
```

# Los strings son arreglos

# Los strings son arreglos

- Como muchos otros lenguajes populares, los strings en Python son arreglos de bytes que representan caracteres unicode, y es posible accesar a sus elementos por medio de los corchetes cuadrados []
- Al igual que los arreglos es posible indexar un substring, y es posible también indexar de forma negativa (no lo hagan).
- Es posible también observar la longitud de string gon el método len

```
b = "Hello, World!"
#imprime un substring
print(b[2:5])
print(b[-5:-2])
#imprime el tamanio del string
print(len(a))
```

# Algunos métodos

# Algunos métodos

- Python tiene varios métodos ya agregados que puedes usar en la clase string

```
a = " Hello, World! "
print(a.strip()) # Regresa "Hello, World!" (remueve los espacios al inicio y
print(a.lower()) # Convierte el string a minusculas
print(a.upper()) # Convierte el string a mayusculas
print(a.replace("H", "J")) # Reemplaza un string por otro
print(a.split(",")) # Regresa ['Hello', ' World!']
x = "hell" in a # Verifica si cierto string se encuentra dentro de otro
print(x)
b = a + a # Concatena 2 strings
```

# Algunos métodos

# Algunos métodos

- Python no permite combinar texto con números (como algunos ya se habrán dado cuenta), por lo que para esto es útil la función format(), y es posible darle una cantidad ilimitada de argumentos

```
age = 20
txt = "My name is Carlos, and I am {}"
print(txt.format(age))
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

- Un diccionario es una colección desordenada, intercambiable e indexada, y es posible accesar a sus elementos con el nombre de su llave dentro de corchetes cuadrados, o con el método get
- Puedes modificar sus valores haciendo referencia a alguna llave

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
x = thisdict.get("model")
thisdict["year"] = 2020
```

# Jugando con diccionarios

- Es posible iterar en un diccionario utilizando el loop for
- Para iterar con el for, se puede acceder a los elementos directamente con corchetes o llaves, co con el método values e items

```
for x in thisdict:
  print(x)
  print(thisdict[x])

for x in thisdict.values():
  print(x)

for x, y in thisdict.items():
  print(x, y)
```

# Más métodos

- Para determinar si un elemento está presente en un diccionario se utiliza la el comando in
- Para determinar cuántos elementos (llave-valor) tiene un diccionario se utiliza len()
- Para agregar un elemento a un diccionario es suficiente agregar una nueva llave y darle un valor

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")

print(len(thisdict))
thisdict["color"] = "red"
print(thisdict)
```

# Removiendo elementos

- Exisen múltiples formas de remover elementos de un diccionario.
- pop() remueve el elemento especificado por la llave
- popitem() remueve el último elemento insertado
- del remueve el item con la llave especificada (igual que pop)
- clear() limpia el diccionario

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
thisdict.popitem()
del thisdict["model"]
thisdict.clear()
```

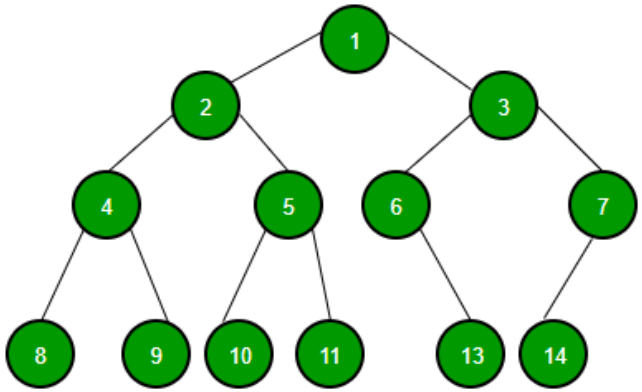
# Copiar un diccionario

- No es posible copiar un diccionario simplemente asignando dict2 = dict1 porque esto crearía solamente una referencia, y un cambio en dict1 se vería reflejado en dict2. Para copiar un diccionario es necesaria la función copy()
- es posible también anidar diccionarios

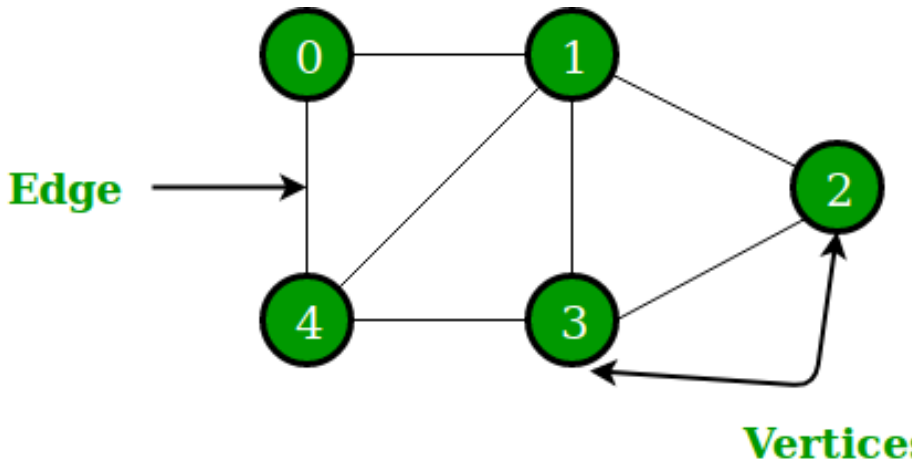
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)

myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

# Árbol binario



# Grafo



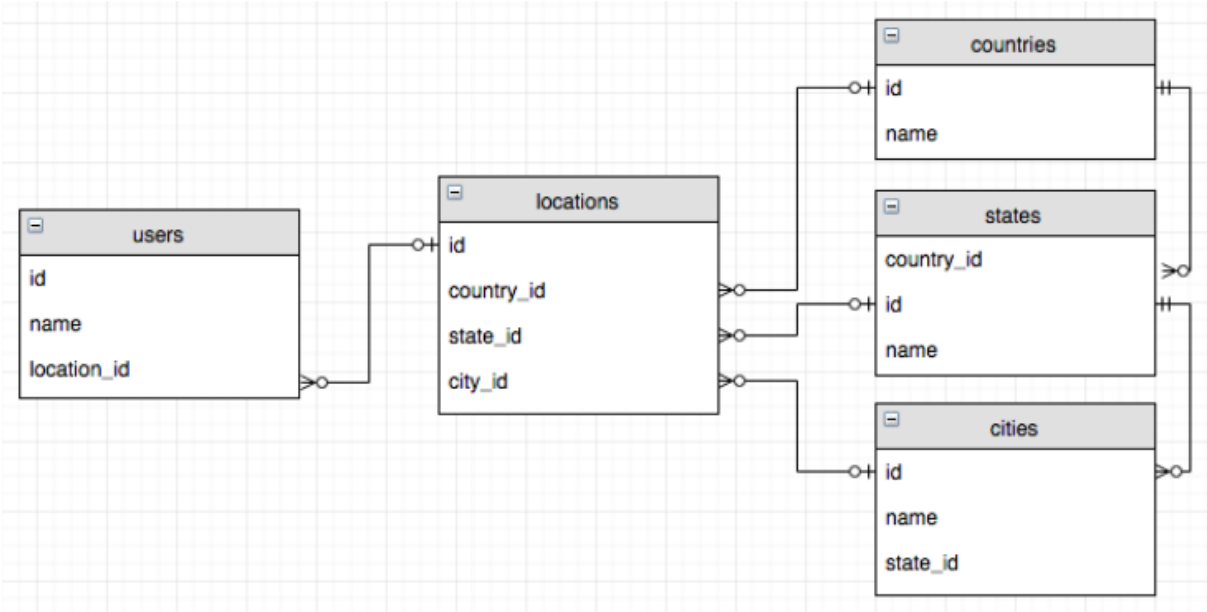
# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# bases de datos relacionales (SQLite, MySQL)

# bases de datos relacionales (SQLite, MySQL)

- Una base de datos relacional se refiere a un objeto digital donde se almacenan datos de una manera tabular en diferentes tablas. Dichas tablas pueden tener relación entre sí. Cada tabla consiste en sets de datos en columnas e hileras.


countries
id
name
locations
id
country_id
state_id
city_id
states
country_id
id
name
users
id
name
location_id
cities
id
name
state_id

# Crear conexión e interactuar con una base de datos (MySQL)

- Para poderse conectar a una base de datos con Python, es necesario tener la librería adecuada para el tipo de base de datos que deseamos.
- Por ejemplo, si deseamos conectarnos a una base de datos como MySQL, podemos hacer uso de la librería llamada mysql-connector-python
- Para instalarlo podemos correr el comando

```
$ pip install mysql-connector-python
```

o

```
$ conda install mysql-connector-python
```

- Una vez instalada la librería, es necesario importarla en nuestro script

```
>> import mysql.connector
>> from mysql.connector import Error
```

- Para crear una conexión se requiere conocer en donde se encuentra alojada la base de datos (host), el nombre de la base de datos a la que queremos acceder (database), el nombre del usuario con el que podemos acceder (use) y por último, la contraseña para dicho usuario (password)

- Para crear una conexión se requiere conocer en donde se encuentra alojada la base de datos (host), el nombre de la base de datos a la que queremos acceder (database), el nombre del usuario con el que podemos acceder (use) y por último, la contraseña para dicho usuario (password)
- En el siguiente bloque de código se intenta crear una conexión a una base de datos local (en nuestro equipo), en el caso que no se logre la conexión, el bloque try, except, finally, lanzará una excepción con el error que se originó al hacer la conexión.

- Si se logra hacer la conexión (se revisa con el método is_connected() del objeto connection), buscamos la información del servidor (con el método get_server_info()), imprimimos dicha información y posteriormente se crea un cursor.

- Si se logra hacer la conexión (se revisa con el método is_connected() del objeto connection), buscamos la información del servidor (con el método get_server_info()), imprimimos dicha información y posteriormente se crea un cursor.
- Un cursor es un puntero que nos permite interactuar con la base de datos.
- Al ejecutar una función, el cursor es quien le indica a la base de datos lo que va a hacer.

- Si se logra hacer la conexión (se revisa con el método is_connected() del objeto connection), buscamos la información del servidor (con el método get_server_info()), imprimimos dicha información y posteriormente se crea un cursor.
- Un cursor es un puntero que nos permite interactuar con la base de datos.
- Al ejecutar una función, el cursor es quien le indica a la base de datos lo que va a hacer.
- En nuestro caso, utilizamos el cursor para ejecutar el comando "select database();" con la función execute.

- Al hacer esto, la base de datos intentará ejecutar los comandos solicitados y los mantendrá en memoria hasta que sean traídos al ambiente de Python.

- Al hacer esto, la base de datos intentará ejecutar los comandos solicitados y los mantendrá en memoria hasta que sean traídos al ambiente de Python.
- Para sacar dicha información podemos hacerlo a través de la función fetchone() o fetchall(). fetchone() iterará cada registro hasta agotar los resultados de la búsqueda, mientras que fetchall() traerá una lista completa con todos los registros.

- En el caso de este script, se seleccionó el nombre de la base de datos a la que se encuentra conectado.

- En el caso de este script, se seleccionó el nombre de la base de datos a la que se encuentra conectado.
- Finalmente, revisamos si existe una conexión, y en caso de que así sea, cerramos el cursor y la conexión.

- (con formato)

```
try:
connection=mysql.connector.connect(host='localhost',
                                   database=prueba,
                                   user='root',
                                   password='')
if connection.is_connected():
    db_Info = connection.get_server_info()
    print("Connected to MySQL Server version ", db_Info)
    cursor = connection.cursor()
    cursor.execute("select database();")
    record = cursor.fetchone()
    print("You're connected to database: ", record)
except Error as e:
    print("Error while connecting to MySQL", e)
finally:
if (connection.is_connected()):
    cursor.close()
    connection.close()
    print("MySQL connection is closed")
```

# Crear conexión e interactuar con una base de datos (Sqlite3)

- Una base de datos Sqlite, en contraste a otras como MySQL, no trabaja sobre un motor de servidor-cliente. Esta base de datos se encuentra contenida en un archivo ordinario, por lo cual tiene limitantes.
- A diferencia también de otras bases de datos, esta no requiere parámetros como usuario y contraseña. Para trabajar con esta, es necesario saber dónde se encuentra el archivo y conectarse a él.

```
try:
    conn = sqlite3.connect('db_sqlite.db')
    cursor = conn.cursor()
    print("Conectado a BD")
except sqlite3.Error as Error:
    print(error)
finally:
    if (conn):
        conn.close()
>>> 347
```

# Seleccionando información

- Ambas bases de datos manejan un lenguaje similar llamado SQL (Structured Query Language) o Lenguaje de Consulta Estructurado. Para seleccionar información de una base de datos es necesario conocer la tabla de la cual queremos buscar información. Si quisiéramos seleccionar todos los registros de una tabla de albums, ejecutamos con un cursor el comando y hacemos un fetchall():

```
query = "SELECT * FROM albums;"
cursor.execute(query)
records = cursor.fetchall()
print(records)
>>> (1, 'For Those About To Rock We Salute You', 1)
>>> (2, 'Balls to the Wall', 2)
>>> (3, 'Restless and Wild', 2)
>>> (4, 'Let There Be Rock', 1)
>>> (5, 'Big Ones', 3)
...
```

- Si queremos sólo ciertas columnas, podemos especificarlas en el SELECT

```
query = "SELECT title FROM albums;"
cursor.execute(query)
records = cursor.fetchall()
print(records)
>>> ('For Those About To Rock We Salute You',)
>>> ('Balls to the Wall',)
>>> ('Restless and Wild',)
>>> ('Let There Be Rock',)
>>> ('Big Ones',)
```

- Existe una cláusula que podemos utilizar para filtrar información, WHERE o dónde, nos permite especificar si queremos que una búsqueda contenga condiciones dentro de la información de sus columnas.

```
query = "SELECT title, artistId FROM albums WHERE artistId > 10 and artistId < 2
cursor.execute(query)
records = cursor.fetchall()
print(records)
[ ('Alcohol Fueled Brewtality Live! [Disc 1]', 11)
('Alcohol Fueled Brewtality Live! [Disc 2]', 11)
('Black Sabbath', 12)
('Black Sabbath Vol. 4 (Remaster)', 12)
('Body Count', 13) ]
```

# Procesando información con Python

- Al ejecutar la función execute() sobre un cursor, podemos obtener las hileras de datos en un solo objeto a través de fetchall(), esto nos regresará una lista con tuplas que representan los datos de cada hilera.

```
query = "SELECT title, artistId FROM albums WHERE artistId > 10 and artistId < 2
cursor.execute(query)
records = cursor.fetchall()
print(records)
[ ('Alcohol Fueled Brewtality Live! [Disc 1]', 11)
('Alcohol Fueled Brewtality Live! [Disc 2]', 11)
('Black Sabbath', 12)
('Black Sabbath Vol. 4 (Remaster)', 12)
('Body Count', 13) ]
```

- Si queremos iterar cada hilera, podemos usar un bucle tipo for:

```
for album in records:
    print(album)
>>> ('Alcohol Fueled Brewtality Live! [Disc 1]', 11)
>>> ('Alcohol Fueled Brewtality Live! [Disc 2]', 11)
>>> ('Black Sabbath', 12)
>>> ('Black Sabbath Vol. 4 (Remaster)', 12)
>>> ('Body Count', 13)
```

- Otra forma de obtener los registros e iterarlos es través de la función fetchone()

```
album = cursor.fetchone()
while album is not None:
    print(album)
    album = cursor.fetchone()
>>> ('Alcohol Fueled Brewtality Live! [Disc 1]', 11)
>>> ('Alcohol Fueled Brewtality Live! [Disc 2]', 11)
>>> ('Black Sabbath', 12)
>>> ('Black Sabbath Vol. 4 (Remaster)', 12)
>>> ('Body Count', 13)
```

# Juntando tablas relacionadas

- Para juntar tablas que tienen relación entre si, podemos usar la función JOIN, y especificamos las columnas que se encuentran relacionadas.

```
query = "SELECT albums.Title, artists.Name FROM albums LEFT JOIN artists on (albu
cursor.execute(query)
records = cursor.fetchall()
for record in records:
        print(record)
>>> ('For Those About To Rock We Salute You', 'AC/DC')
>>> ('Balls to the Wall', 'Accept')
>>> ('Restless and Wild', 'Accept')
>>> ('Let There Be Rock', 'AC/DC')
>>> ('Big Ones', 'Aerosmith')
```

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Instalando e Importando NumPy

# Instalando e Importando NumPy

- Numpy es una librería que nos permite trabajar con datos de una manera muy eficiente y rápida.

```
$pip install numpy, o conda install numpy
```

```
import numpy as np
```

# Creando arreglos y matrices

- Crear arreglo de np de una lista de Python

```
lista = [1, 2, 3, 4, 5]
lista_np = np.array(lista)
>>> array([1, 2, 3, 4, 5])
```

- Con valores inicializados en 0 (np.zeros())

```
np_lista_zeros = np.zeros(10)
np_lista_zeros
>>> array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

# Creando arreglos

- Con valores inicializados en 1 (np.ones())

```
np_lista_ones = np.ones(10)
np_lista_ones
>>> array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

- Definiendo un comienzo, fin y la cantidad de elementos (np.linspace())

```
np_linspace = np.linspace(1, 10)
print(len(np_linspace))
>>> 50
np_linspace = np.linspace(1, 10, 10)
print(np_linspace)
>>> [ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

# Creando arreglos

- Con valores aleatorios (np.random.randint())

```
np_random = np.random.randint(10, size=10)
np_random
>>> array([7, 8, 4, 3, 4, 7, 7, 2, 6, 2])
np_random = np.random.randint(10, 20, size=10)
np_random
>>> array([15, 11, 17, 16, 19, 16, 19, 10, 19, 19])
```

# Creando matrices

- Para crear matrices de 2 o más dimensiones usamos la función deseada para los valores default (zeros(), ones(), etc), y en vez de pasar el número de elementos que queremos, pasamos una tupla especificando la forma que queremos.
- Para una matriz de 2 dimensiones pasamos (x, y), para una de 3 pasamos (x, y, z), para más niveles (x, y... n)

# Creando matrices

```
np.zeros((4, 4))
>>> array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
np.zeros((3, 3, 3))
>>> array([[[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]]])
```

# Manipulación y operaciones en arreglos

- Cómo obtener secciones de arreglos

```
lista = [1, 2, 3, 4, 5]
lista_np = np.array(lista)
>>> array([1, 2, 3, 4, 5])
print(lista_np[0])
>>> 1
print(lista_np[0:2])
>>> [1, 2]
print(lista_np[:4])
>>> [1, 2, 3, 4]
print(lista_np[-1]
>>> 5
print(np.where(lista_np > 2))
>>> [1, 2, 3, 4] # regresa el index
nueva_lista = lista_np[np.where(lista_np > 2)]
print(nueva_lista)
>>> [3, 4, 5]
```

# Manipulación y operaciones en arreglos

- Operaciones y funciones de arreglos

```
np.sum(lista_np)
>>> 15
np.prod(lista_np)
>>> 120
np.mean(lista_np)
>>> 3
np.std(lista_np)
>>> 1.4142135623730951
np.var(lista_np)
>>> 2.0
np.min(lista_np)
>>> 1
np.max(lista_np)
>>> 5

lista_np + lista_np
>>> [2,  4,  6,  8, 10]
lista_np + 30
>>> [31, 32, 33, 34, 35]
lista_np * lista_np
>>> [ 1,  4,  9, 16, 25]
```

# Manipulación y operaciones en arreglos

- Operaciones con np.where()

```
np.where(lista_np > 2, 10, lista_np)
>>> [ 1,  2, 10, 10, 10]
np.where(lista_np > 2, 10, 0)
>>> [ 0,  0, 10, 10, 10]
```

# Manipulación y operaciones en arreglos

- Cambiar la figura/aspecto del arreglo (np.shape(), np.reshape(), np.flatten())

```
lista_np2 = np.array([1,2,3,4,5,6,7,8,9,10])
lista_np2.shape = (5,2)
print(lista_np2)
>>> [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]]
lista_np2.shape = (2, 5)
print(lista_np2)
>>> [[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
print(lista_np2.flatten())
>>> [ 1  2  3  4  5  6  7  8  9 10]
```

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Ejercicios

# Ejercicios

- Escribe un programa que regrese un string hecho por los primeros 2 y los últimos 2 caracteres de un string dado. Si la longitud del string es menor a 2, regresar un string vacío. (ejemplo: Entradag : 'python3' Salida: 'pyn3'
- Dado un string, crear una función que cuente la ocurrencia de cada letra dentro del string
- Implementar una función sumatoria de 2i+1 donde tenemos una lista de valores numéricos, los cuales multiplicaremos cada valor por 2, seguido le sumaremos un 1 a cada valor, y finalmente sumaremos todos los valores de la lista lista = [1, 2, 3 ,4] Resultado = 24
- Escribe un programa que remueva los duplicados de un diccionario
- Escribe un programa que escriba en un archivo los valores de un diccionario separando la llave de su valor en diferentes columnas
- Escribe un programa que calcule el determinante de un arreglo

# Ejercicios

# Ejercicios

- Crear una función que se conecte al archivo de sqlite, buscar todos los albums y juntarlos con la tabla de artists. Iterar todos los registros y crear un diccionario donde la llave será el nombre del artista y su contenido será un arreglo con la lista de sus albums

**ejemplo:**

```
{
    'AC/DC': [
    'For those about to Rock we salute You',
    'Let There Be Rock'
    ],
'Accept': []
}
```

# Outline


1. Comenzando el viaje a al programación
2. Aterrizando a python
3. Tipos de datos (otra vez)
4. Interactuando con el usuario
5. Tipos de datos (última continuación)
6. Conjuntos
7. Indentación (sangría) en python
8. Control de flujo y condicionales
9. Módulos y Funciones
10. Archivos I/O
11. OOP
12. Strings
13. Diccionarios
14. Bases de datos
15. Introducción a Numpy
16. Ejercicios (Día 2)
17. Bibliografía

# Referencias

- (=

# Referencias

- (=

## Bibliografías


https://www.tutorialspoint.com/python/python_basic_ope
https://www.tutorialsteacher.com/python/python-compari
https://www.learnpython.org/es/Hello,%20World!
https://www.w3schools.com/python/python_lists.asp
https://www.guru99.com/reading-and-writing-files-in-py
https://www.tutorialpython.com/modulos-python/
https://www.tutorialspoint.com/python_data_structure/p