|  |  |
| --- | --- |
| **1)**   * Dinámico y fuertemente tipado. * Se concatena con el signo **+**. * Para imprimir números hay que convertirlo con **str(número)**. * Escribir un mandato en varias líneas con **\**. * Se comenta con **#**. * Los tipos de datos están en figura 1. * Los operadores están en figura 2. * Todo es un objeto. * Para ver el tipo de dato: **type(variable)**. * Las variables son válidas en su ámbito. * Referencias: <http://pyspanishdoc.sourceforge.net/> | **2) Funciones**:   * Su sintaxis es:   **def <nombre\_funcion>([param1, param2, param3, …]):**  **<tabulación/4 espacios> instrucciones…**  **<tabulación/4 espacios> [return [resultado]]** |
| **3) Listas**:  listaUno = ["Juan", 3, True, False, 0.12, "Ana"]  listaDos = ["Pedro", 2, 5, 4.435, False]  listaTres = []  Formas de imprimir:  print(listaUno[:])  # ['Juan', 3, True, False, 0.12, 'Ana']  print(listaUno[:3])  # ['Juan', 3, True]  print(listaUno[2:])  # [True, False, 0.12, 'Ana']  print(listaUno[-3:])  # [False, 0.12, 'Ana']  print(listaUno[:-3])  # ['Juan', 3, True]  print(listaUno[1:5])  # [3, True, False, 0.12]  Agregar un elemento al final:  listaUno.**append**("Manuel")  Agregar elemento en otra posición:  listaUno.**insert**(3, "Daniela")  Agregar varios elementos:  listaUno.**extend**([2, 10, "Gary"])  Conocer índice del primer elemento encontrado:  indiceElemento = listaUno.**index**("Manuel")  Buscar elemento en una lista:  estaElemento = "Manuel" **in** listaUno  Eliminar un elemento:  listaUno.**remove**(False)  Eliminar último elemento:  listaUno.**pop()**  Unir dos listas:  listaTres = listaUno **+** listaDos  Repetir elementos de una lista:  listaCuatro = [1, 2, 3] **\*** 4 | **4) Tuplas**:   * No se pueden modificar. * No permite búsquedas. * Se coloca entre paréntesis.   miTupla = ("Juan", 13, 13, 1, 1995)  miTupla4 = ("juan", 13, 1, 1995)  miLista = ["Diego", 22, 44, 3333]  Pasar de tupla a lista:  lista = **list**(miTupla)  Pasar de lista a tupla:  tupla = **tuple**(miLista)  Ver si un elemento está en una tupla:  esta = "Juan" **in** miTupla  Ver cuantas veces se repite un elemento en una tupla:  cuantasVeces = miTupla.**count**(13)  Ver cuántos elementos hay:  cuantosElementos = **len**(miTupla)  Tupla unitaria, debe ir con coma al final:  miTupla2 = ("Juan"**,**)  Las tuplas pueden ir sin paréntesis:  miTupla3 = "Juan", 13, 13, 14, 1, 1995  Desempaquetado de tuplas:  nombre, dia, mes, anho **=** miTupla4 |
| **5) Diccionario**:   * Se usa la asociación **clave:valor**. * Va entre llaves (**{}**).   miDiccionario = {  "Alemania":"Berlín",  "Francia":"París",  "Reino Unido":"Londres",  "España":"Madrid"  }  miDiccionario2 = {  "Alemania": 1945,  4:"Jordan",  1979:"Manuel"  }  Agregar una nueva clave:valor:  miDiccionario["Italia"] = "Lisboa"  Modificar una clave:valor:  miDiccionario["Italia"] = "Roma"  Eliminar una clave:valor:  **del** miDiccionario["Reino Unido"]  Asignar valores de una tupla a un diccionario:  miTupla=(  "España",  "Francia",  "Reino Unido",  "Alemania"  )  miDiccionario3 = {  miTupla[0]:"Madrid", miTupla[1]:"París", miTupla[2]:"Londres", miTupla[3]:"Berlín"  }  Diccionario con una tupla como valor:  miDiccionario4 = {  23:"Jordan",  "Nombre":"Michael",  "Equipo":"Chicago",  "Anillos":(  1991,  1992,  1993,  1996,  1997,  1998  )  }  Diccionario como un diccionario y una tupla como valor:  miDiccionario5 = {  23:"Jordan",  "Nombre":"Michael",  "Equipo":"Chicago",  "Anillos":{  "temporadas":(  1991, 1992, 1993,  1996, 1997, 1998  )  }  }  Obtener las claves:  print(miDiccionario.**keys()**)  Obtener los valores:  print(miDiccionario.**values()**)  Obtener la longitud:  print(**len**(miDiccionario)) | **6) Condicionales**:  **IF**:  if <condición>:  <tabulación> instrucciones 1…N  elif <condición>:  <tabulación> instrucción 1…N  else:  <tabulación> instrucción 1…N  **SWITCH**: No existe, pero se puede usar IF con,   * Concatenar operadores de comparación. * Operadores lógicos AND y OR. * Operador IN   **Lower()**: pasa a minúscula.  **Uppper()**: pasa a mayúscula.  **FOR**:  for variable in [elemento]:  <tabulación> instrucciones 1…N  <tabulación> [break | return | continue | pass | else]  for variable in range(cantidad):  <tabulación> instrucciones 1…N  <tabulación> [break | return | continue | pass | else]  **Range(cantidad)**: recorreo el for una cantidad de elementos como un arreglo.  **Print(“”, end=” ”)**: imprime sin salto de línea.  **Print(“”, sep=”separador”)**: imprime con un separador.  **WHILE**:  while <condición>:  <tabulación> instrucciones 1…N  <tabulación> [break | return | continue | pass | else]  **Math.sqrt()**: calcula raíz cuadrada, hay que importar librería math.  **Len(elemento)**: cantidad de elementos.  **While True**: ciclo infinito. |
| **7) GENERADORES**:   * Extrae valores de una función y se almacenan en objetos iterables que se pueden recorrer. * Los valores se almacenan de uno en uno. * Cada vez que un generador almacena un valor, permanece en un estado pausado hasta que se solicita el siguiente (suspensión de estado).   Ventaja:   * Más eficiente que las funciones tradicionales. * Muy útiles con listas de valores infinitos.   Sintaxis:  def <nombre generador>():  <tabulación> yield <elemento devolver>  <tabulación> [return]  Ejemplo:    Donde:   1. **Next**: llama para obtener el próximo elemento iterativo.  * **Yield from**, simplifica el código de los generadores en el caso de utilizar bucles anidados.   **\*parámetro**: si el parámetro tiene un asterisco, quiere decir que se envía una tupla con N elementos.    Ejemplo: | **8) EXCEPCIONES**:   * Son errores que ocurren durante la ejecución del programa, la sintaxis es correcta, pero algo ocurre. * Si se escribe **except** solo, toma cualquier excepción. * La instrucción **Raise** permite personalizar la excepción.   Sintaxis:  **try:**  **pass**  **except Exception as e:**  **raise**  **else:**  **pass**  **finally:**  **pass** |

|  |  |
| --- | --- |
| **9) POO**:   * Encapsular permite proteger una variable, a través de doble guion bajo. Queda accesible solo dentro de la clase.   Ejemplo:     * Encapsular un método es similar, y quiere decir que solo se puede acceder a el desde dentro de la propia clase. * Herencia, permite reutilizar objetos similares. * Cuando hay herencia múltiple, se da prioridad a la primera clase que hereda. * Para tener una clase vacía y el código no arroje error se usa la palabra clave **pass**.   Ejemplo:     * **Super()**: Permite acceder al constructor y metodos de la clase padre. * **Principio de sustitución**: es siempre un/una, quiere decir, que una subclase siempre es un elemento de la clase padre, ejemplo, un empleado siempre es una personal, pero una persona no necesariamente es un empleado. * **Isinstance()**: Verifica si un objeto es instancia de una clase. Devuelve True o False.   Ejemplo:     * **Polimorfismo**: indica que un objeto puede cambiar de forma, dependiendo del contexto.   Ejemplo:    **14) ARCHIVOS EXTERNOS**:   * Permite la persistencia de datos. * Las fases para guardar información son:   + Creación.   + Apertura.   + Manipulación.   + Cierre. * Ayuda externa, <https://docs.python.org/3/library/io.html> * Los modos son lectura (r), escritura (w) y añadir (a). * Se debe importar la clase **open** de la librería **io**. * **W** y **a** crear el fichero si este no existe. * Con la opción **seek(puntero)**, se indica la posición donde se comenzará a leer.   Ejemplo escritura:    Ejemplo lectura:    Ejemplo Lectura en lista:    Ejemplo añadir:    Ejemplo seek(): | **10) METODOS DE CADENA**:   * **Upper()**: Pasa a mayúscula. * **Lower()**: Pasa a minúscula. * **Capitalize()**: Pasa la primera letra en mayúscula. * **Count()**: Cantidad de veces que aparece una carácter o cadena dentro de un texto. * **Find()**: Indica la posición donde aparece un carácter o grupo de caracteres dentro de un texto. * **Isdigit()**: True/False si un valor es número o no. * **Isalum()**: Comprueba si es alfanumérico. * **Isalpha()**: Comprueba si hay solo letras. No cuenta los espacios. * **Split()**: Separa por palabras utilizando espacios. * **Strip()**: Borra espacios al principio y al fin. * **Replace()**: Cambia palabra o letra por otra. * **Rfind()**: Lo mismo que find contando desde atrás.     **11) MODULOS**:   * Son archivos con extensión PY, PYT (Python compilado) o escritos en C para CPython, que poseen su propio espacio de nombres y que pueden contener variables, funciones, clases e incluso otros módulos. * Sirve para organizar y reutilizar código (modularización y reutilización). * Para usar el módulo hay que importarlo, se puede hacer con:   **import <módulo>**.   * La forma recomendable para usar un módulo es:   **from <módulo> import <clase/función>**  **from <módulo> import \***  **12) PAQUETES**:   * Son directorios donde se almacenarán módulos relacionados entre sí. * Sirve para organizar y reutilizar los módulos. * Se crea una carpeta con un archivo **\_\_init\_\_.py** * Se utiliza de la forma siguiente:   **from <capeta>.<fichero> import <clase/función>**  **from <capeta>.<fichero> import \***   * Se pueden crear sub paquetes dentro de varias, solo hay crear e archivo \_\_init\_\_.py dentro de cada carpeta.   **13) PAQUETES DISTRIBUIBLES**:   * Genera un instalador. * Permite que este donde este se puede llamar al paquete. * Para generar un paquete, se hace:   + Generar archivo **setup.py**.   + Agregar el código siguiente:      * + En la misma carpeta de la Usar por DOS el mandato:   **python setup.py sdist**   * + Se van a generar 2 carpetas, ingresar a la llamada **dist**.   + Ingresar a la carpeta **dist** por consola y ejecutar:   **pip3 install <nombre\_paquete>**   * + Lo anterior instalara el paquete.   + Para desinstalar el paquete usar el mandato:   **pip3 uninstall <nombre\_paquete>** |

Diagrama

Descripción generada automáticamente

**Figura 1. Tipos de datos.**

Diagrama

Descripción generada automáticamente

**Figura 2. Operadores de datos.**

https://www.youtube.com/watch?v=Ufqh8aoR9hE&list=PLU8oAlHdN5BlvPxziopYZRd55pdqFwkeS&index=8