Funciones, listas, diccionarios, archivos

Repaso de los conceptos utilizando Python

Funciones y procedimientos

La definición de una función requiere especificar el nombre de ella y la secuencia de pasos que se ejecutarán al invocarla.

```
def mi_primera_funcion():
    print("Esta es mi primera función en Python.")
    print(":D")
```

Funciones y procedimientos

- **def** es una palabra reservada que indica que es una declaración de función.
- El nombre de la función es mi_primera_funcion.
  - Las reglas de nombres para funciones son las mismas que para variables.
- Los paréntesis vacíos significan que la función no acepta parámetros de entrada.
- Una vez definida, la función puede usarse en cualquier lugar donde puede ir una sentencia.

Parámetros

```
def saludar(nombre):
    print("Hola " + nombre + "!!")
```

- Al momento de invocar a la función, debemos proveer **un valor para cada parámetro de la función**.
- Ese valor puede ser un literal, una expresión, una variable, otra invocación a función, etc...

```
saludar("Pedro"  + "Garcia")
saludar(input("Ingrese su nombre: "))
```

Funciones. Retorno de valores

Existen funciones que no retornan ningún resultado. (procedimientos)

- Ej: La función **print()** muestra una cadena por pantalla pero no retorna ningún resultado después de su ejecución.

Otras funciones SI retornan un valor de resultado.

- Ej: La función **min()** retorna el menor elemento de una secuencia que recibe como parámetro.

Funciones. Retorno de valores

Para retornar un valor de una función definida por nosotros, usaremos la palabra reservada **return**.

```
def sumar(a, b):
    total = a + b
    return total

res = sumar(4,7)      # res vale 11
```

Tipos de datos en funciones

- Podemos indicar el tipo de dato que queremos recibir, aunque python no controlará que se respete.
- También podemos indicar el tipo de dato que retornará la función.

```
def saludar(nombre:str):
    print("Hola " + nombre + "!!")

def sumar(a:int, b:int)->int:
    return a + b
```

Recordando...

la modularización de un programa en funciones es muy recomendable por:

- El código es más fácil de leer, entender, modificar, etc...
- Elimina la duplicación de código. Si un código se repite en diferentes partes de un programa, debo pensar en hacer una función con él.
- Dividir un programa en subprogramas (funciones) permite disminuir la complejidad e ir resolviendo “por partes”.
- Las funciones bien diseñadas pueden usarse, inclusive, en distintos programas. Ej: Librerías

Listas / arrays

Las listas en Python son un tipo de dato que permite almacenar datos de cualquier tipo. Son **mutables** y **dinámicas**, lo cual es la principal diferencia con los *sets* y las *tuplas*.

Permiten almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas prácticamente lo que sea.

```
lista = [1, 2, 3, 4]
```

Listas. Propiedades

- Son ordenadas, mantienen el orden en el que han sido definidas
- Pueden ser formadas por tipos arbitrarios

- Pueden ser indexadas con **[i]**.
- Se pueden anidar, es decir, meter una dentro de la otra.
- Son mutables, ya que sus elementos pueden ser modificados.
- Son dinámicas, ya que se pueden añadir o eliminar elementos.

Listas

Para crear una lista:

```
lista_vacia = list()
lista_vacia = []
lista_enteros = [1, 3, 900]
lista_heterogenea = ["Hola", 1, 2.34]
lista_de_listas = ["Elemento", [1, 2, 3], [3.4,
"casa"]]
```

Listas

```
lista_enteros = [1, 3, 900]
lista_de_listas = [“Elemento”, [1, 2, 3], [3.4, “casa”]]

print(lista_enteros[1]) ⟶  # Salida: 3
print(lista_de_listas[2]) ⟶  # Salida: [3.4,“casa”]

lista_enteros[1] = 1000
print(lista_enteros[1]) ⟶  # Salida: 1000
```

Listas. Indexado

Debemos controlar que el índice exista, de lo contrario tendríamos un error en la ejecución (excepción de índice fuera de rango).

Si el índice es negativo, se cuenta desde el final de la lista.

```
lista_enteros = [1, 3, 900, 2000, 5, 4]

print(lista_enteros[-1])  # Salida: 4
```

Pertenencia

Para evaluar si un elemento está en una lista, usamos el operador **in**.

```
lista = [1, 4, 5, "palabra"]
print(1 in lista)
print(2 in lista)
print("palabra" in lista)
```

Salida:

True

False

True

Listas. Recorrida

- Para leer los elementos de una lista, puedo usar el ciclo for.

```
personas = ["José", "María", "Ana", "Pedro"]
for amigo in personas:
        print(amigo)
```

- Para modificar los elementos de una lista, debo conocer la posición de cada elemento:

```
for i in range(len(numbers)):
        numbers[i] = numbers[i] * 2
```

Operaciones sobre listas

Análogamente a strings, el símbolo + concatena dos listas.

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
```

salida:

[1, 2, 3, 4, 5, 6]

Slicing

Es posible crear sublistas más pequeñas de una más grande. Para ello debemos de usar **:** entre corchetes, indicando a la izquierda el valor de inicio, y a la izquierda el **valor final que no está incluido**. Por lo tanto **[0:2]** creará una lista con los elementos **[0]** y **[1]** de la original.

```
lista = [1, 2, 3, 4, 5, 6]
print(lista[0:2])   # salida [1, 2]
print(lista[2:6])   # salida [3, 4, 5, 6]
lista[0:3] = [0, 0, 0]
print(lista)        # salida [0, 0, 0, 4, 5, 6]
```

Slicing

La asignación de una lista a otra variable genera una **referencia** al mismo espacio en memoria, por lo tanto los cambios que hagamos a la segunda variable se verán impactados en la lista original.

`aux = lista` (en este caso aux es una referencia a lista, los cambios que realicemos sobre `aux` impactan a la lista original `lista`)

Para hacer una **copia independiente** de una lista debemos hacerlo con **slice** o con el método **copy()**:

```
aux = lista[:]
```

```
aux = lista.copy()
```

Ejemplo copia de lista

```
lista = [1, 2, 3, 4, 5, 6]

aux = lista

print("aux:", aux)

aux.pop(0)

aux.pop(1)

print("aux post cambios:",aux)

print("lista post cambios:",lista)
```

Salida
aux: [1, 2, 3, 4, 5, 6]
aux post cambios: [2, 4, 5, 6]
lista post cambios: [2, 4, 5, 6]

Ejemplo copia de lista

```
lista = [1, 2, 3, 4, 5, 6]

aux = lista[:]

print("aux:", aux)

aux.pop(0)

aux.pop(1)

print("aux post cambios:",aux)

print("lista post cambios:",lista)
```

Salida
aux: [1, 2, 3, 4, 5, 6]
aux post cambios: [2, 4, 5, 6]
lista post cambios: [1, 2, 3, 4, 5, 6]

Métodos de listas

- **.append(elem)** inserta elementos al final de una lista.
- **.insert(pos, elem)** inserta un elemento en la posición **pos**.
- **.index(elem)** devuelve la posición de un elemento, si no está produce un error.
- **.pop(pos)** quita el elemento de la posición **pos** de la lista.
- **.remove(elem)** recibe como argumento un elemento y lo borra de la lista, si no existe produce un error.
- **.sort()** ordena los elementos de menor a mayor por defecto. Y también permite ordenar de mayor a menor si se pasa como parámetro `reverse=True`.
  ejemplo: **lista.sort(reverse=True)**

.append(elemento)

```
lista=[4, 100, 54, 23, 78]
print(lista, ", Cantidad de elementos:", len(lista))
lista.append(200)
lista.append(143)
print(lista, ", Cantidad de elementos:", len(lista))
```

Salida:

[4, 100, 54, 23, 78] Longitud: 5

[4, 100, 54, 23, 78, **200, 143**] Longitud: 7

.insert(pos, elemento)

```
lista.insert(0,1.3)
lista.insert(1,2.5)
lista.insert(2,"Los anteriores fueron insertados")
print(lista)
print("Cantidad de elementos:", len(lista))
```

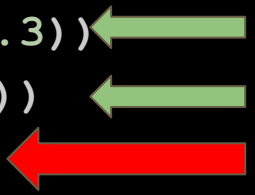
Salida:

[1.3, 2.5, 'Los anteriores fueron insertados', 4, 100, 54, 23, 78, 200, 143]

Cantidad de elementos: 10

.index(elem)

```
lista = [1.3, 2.5, 'Los anteriores fueron insertados', 4, 100,
54, 23, 78, 200, 143]
print("indice de 1.3:", lista.index(1.3))
print("indice de 78:", lista.index(78))
print("indice de 2:", lista.index(2))
```



Salida:

```
indice de 1.3: 0
indice de 78: 7
Traceback (most recent call last):
  File "c:\Damian\ … \Programa.py", line 58, in <module>
    print("indice de 2:",lista.index(2))
                         ^^^^^^^^^^^^^^
ValueError: 2 is not in list
```

.pop(pos)

```
lista = [1.3, 2.5, 4, 100, 54, 23, 78, 200, 143]
print("Elemento en pos=7 :", lista[7])
lista.pop(7)
print("Elemento en pos=7 :", lista[7])
print("Lista actual :", lista)
```

Salida

Elemento en pos=7 : 200

Elemento en pos=7 : 143

Lista actual : [1.3, 2.5, 4, 100, 54, 23, 78, 143]

Try-Except

El bloque **try** se utiliza para manejar errores que pueden ocurrir durante la ejecución de un programa. Es como decirle al programa: "*Intenta ejecutar este código, y si algo sale mal, no te detengas; en su lugar, haz algo diferente para manejar el error*". Componentes:

1. **try**: Aquí colocas el código que quieres ejecutar. Si todo va bien, el programa sigue su curso normal.
2. **except**: Si ocurre un error dentro del bloque try, el programa salta al bloque except y ejecuta el código que hayas colocado allí. De esta manera, evitas que el programa se detenga abruptamente por un error.
3. **else (opcional)**: Este bloque se ejecuta solo si no ocurrió ningún error en el bloque try.
4. **finally (opcional)**: Este bloque se ejecuta al final, sin importar si hubo un error o no. Es útil para liberar recursos o realizar tareas de limpieza.

Para resolver en clase

Tomémonos unos minutos para pensar la solución a los siguientes problemas con lo visto hasta el momento:

1. Generar una lista con los elementos pares múltiplos de 3, menores a un número ingresado por el usuario. Dividir el problema en subproblemas.

```
def leer_entero(mensaje:str)->int:
    repetir = True
    while repetir:
        try:
            ingreso = input(mensaje)
            numero = int(ingreso)
            repetir=False
        except ValueError:
            print('Error, debe ingresar un número entero.')
                  print('Inténtelo nuevamente:')
    return numero
```

```
lista_pares=[]
limite = -1
while limite < 0:
    limite = leer_entero("Ingrese un número entero
positivo: ")

for i in range(0, limite):
    if i%3==0 and i%2==0:
        lista_pares.append(i)

print(lista_pares)
```

Para resolver en clase

2. Tenemos cargada una lista con las notas del primer parcial.
El profesor decide cambiar el puntaje de un punto, y eso implica darles 5 puntos más a cada nota porque todos tenían bien ese punto.

Ejemplo:

notas = [80, 60, 75, 100, 55, 35]

notas_ajustadas = [85, 65, 80, 100, 60, 40]

Soluciona?

```
notas = [80, 60, 75, 100, 55, 35]
notas_ajustadas = notas
for i in range(len(notas_ajustadas)):
    if notas_ajustadas[i] > 95:
        notas_ajustadas[i] = 100
    else:
        notas_ajustadas[i] += 5
print("notas ajustadas", notas_ajustadas)
print("notas originales", notas)
```

notas_ajustadas es una referencia a la lista notas, no es una copia independiente

¿Por qué?

Salida:

notas ajustadas [85, 65, 80, 100, 60, 40]

Notas originales [85, 65, 80, 100, 60, 40]

Corrección

```
notas = [80, 60, 75, 100, 55, 35]
notas_ajustadas = notas[:]
for i in range(len(notas_ajustadas)):
    if notas_ajustadas[i] > 95:
        notas_ajustadas[i] = 100
    else:
        notas_ajustadas[i] += 5
print("notas ajustadas", notas_ajustadas)
print("notas originales", notas)
```

Otra forma válida:
`notas_ajustadas = notas.copy()`

Salida:
notas ajustadas [85, 65, 80, 100, 60, 40]
notas originales [80, 60, 75, 100, 55, 35]

Diccionarios

Las listas indexan sus elementos a partir de sus posiciones:

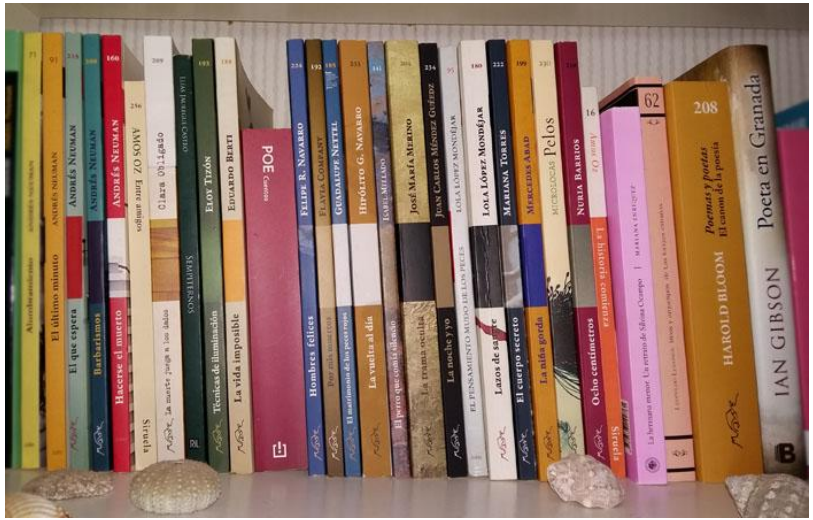
```
l = [‘a’, ‘b’, ‘c’]

l[1] // -> ‘b’
```

Los **diccionarios** son “bolsas”, sus elementos no están ordenados.

La forma de indexar elementos en un diccionario es a través de su clave.

Diccionarios

Listas



Secuencia de elementos

Diccionarios

“Bolsa” de elementos etiquetados

Diccionarios

```
mochila = dict()
mochila = {}

mochila["llaves"] = 2

mochila['auriculares'] = 1

mochila["caramelos"] = 20

print(mochila)

# {'llaves': 2, 'auriculares': 1, 'caramelos': 20}
```

2 formas de crear un diccionario vacío

Diccionarios

```
mochila['caramelos'] += 10

print(mochila['caramelos'])

# 30

print(mochila['billetera'])

KeyError: 'billetera'
```

Listas vs Diccionarios

```
l = list() # o l=[]
l.append(“Juan”)
l.append(33)
print(l)
['Juan', 33]
print(l[0])
'Juan'
```

```
d = dict() # o d={}
d[“nombre”] = “Juan”
d[“edad”] = 33
print(d)
{'nombre': 'Juan', 'edad': 33}
print(d[“nombre”])
'Juan'
```

Operador in

Como dijimos anteriormente, es un error hacer referencia a una clave que no existe.

Por lo tanto, debemos poder verificar si una dada clave existe o no en un diccionario.

Eso se hace con el operador de pertenencia **in**. (igual que en las listas)

```
print('llaves' in mochila)     #  True

print('billetera' in mochila)   #  False
```

Iteración sobre diccionarios

Si bien los diccionarios no mantienen un orden sobre los elementos, podemos usar el ciclo for para recorrer cada uno de los elementos.

No podemos asumir el orden del recorrido!!

El ciclo for recorrerá el diccionario a partir de las claves:

```
for clave in mochila:
    print(clave)
```

Salida:
llaves
auriculares
caramelos

Claves-Valores

Podemos obtener listas con las claves y valores de un diccionario:

```
cuenta = {"pesos": 13423, "dolares": 2345, "euros":1990}

print(cuenta.keys())
# salida: dict_keys(['pesos', 'dolares', 'euros'])

print(cuenta.values())
# salida: dict_values([13423, 2345, 1990])

print(cuenta.items())
# salida: dict_items([('pesos', 13423), ('dolares',
2345), ('euros', 1990)])
```

Iteración de dos variables

Utilizando la noción de tupla, podemos expresar un ciclo for con dos variables para recorrer un diccionario.

```
for moneda, cantidad in cuenta.items():

    print(f"Tenes {cantidad} {moneda}")
```

Tenes 13423 pesos

Tenes 2345 dolares

Tenes 1990 euros

En cada iteración moneda es la **clave** de la entrada y cantidad su **valor**

lista.items() devuelve una lista de tuplas en la forma **[(clave1, valor1), (clave2, valor2), ...]**.

El bucle for recorre estas tuplas, desempaquetando cada una en las variables clave y valor, lo que te permite acceder a la clave y al valor de cada par en el diccionario.

Archivos

Antes de leer el contenido de un archivo en Python, tenemos que explicitar con qué archivo vamos a trabajar y que es lo que haremos con el (leer o escribir).

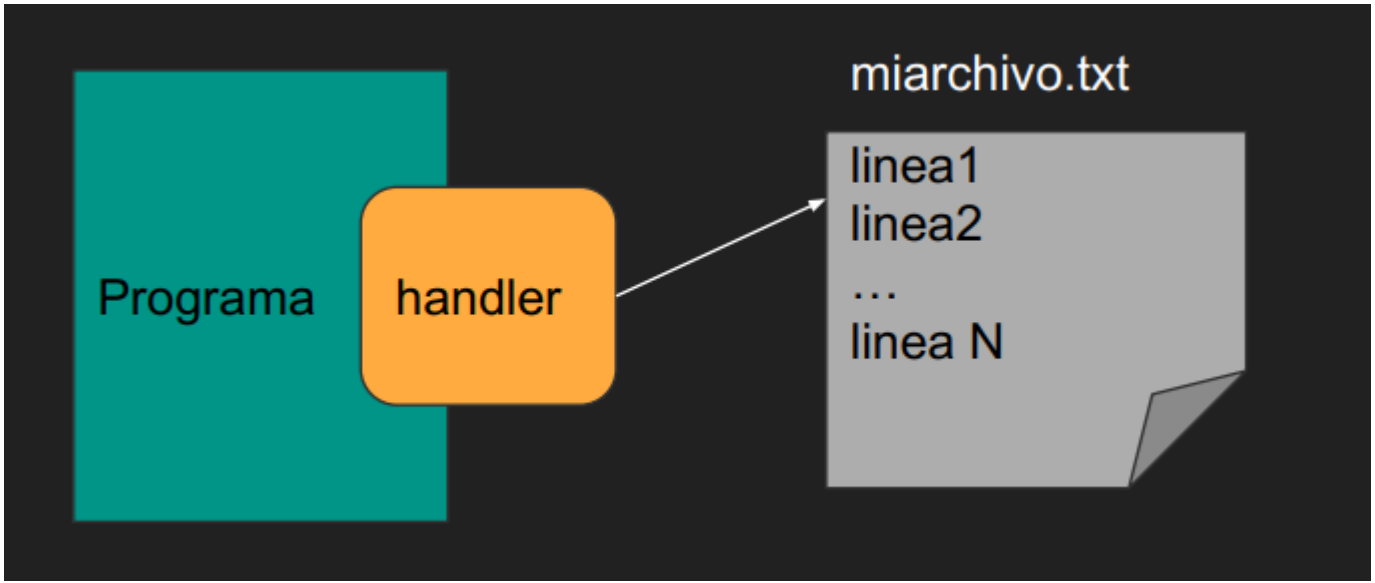
Esto se realiza con la función **open** (predefinida en el lenguaje, no la tenemos que implementar nosotros).

La función open NO lee el contenido del archivo, solamente nos provee un mecanismo para acceder a él (**handler**)

Archivos

Sintaxis: **handle = open(nombre, modo)**

Ejemplo: `arch = open("miarchivo.txt", "r")`



Archivos

handle = open(nombre, modo)

El primer argumento es una cadena que contiene el nombre del fichero.

El segundo argumento es otra cadena que describe la forma en que el fichero será usado. **modo** puede ser:

- **'r'** cuando el fichero solo se leerá
- **'w'** para sólo escritura (un fichero existente con el mismo nombre se borrará)
- **'a'** abre el fichero para agregar; cualquier dato que se escribe en el fichero se añade automáticamente al final
- **'r+'** abre el fichero tanto para lectura como para escritura.

El argumento modo es opcional; se asume que se usará 'r' si se omite.

Leer un archivo de texto

Abrir un archivo no existente es un error.

El handle del archivo puede usarse como secuencia de strings con el contenido del archivo.

Entonces podemos usar un ciclo for para iterar sobre ella.

```
archivo = open(“miarchivo.txt”, ‘r’)
for linea in archivo:
        print(linea)
```

Ejercicios

1. Implementar un programa para contar las líneas de un archivo.
2. Implementar un programa para contar los caracteres de un archivo.
3. Dado un archivo de la forma:

```
43;aspiradora;30
15;tostadora;20
20;cafetera;0
....
```

Implementar un lector para cargar esa información en una lista de diccionarios de la forma {“codigo”: ..., “producto”:..., “cantidad”: ...}. Investigar la operación split para separar por un caracter determinado (no espacio).

1

```
ruta = r"c:\Programacion2\productos.txt"
archivo = open(ruta, "r")
contador = 0
for linea in archivo:
    contador += 1
print("Lineas:", contador)
archivo.close()
```

1 - alternativa:

```
ruta = r"c:\Programacion2\productos.txt"
archivo = open(ruta, "r")
lista_lineas = archivo.readlines()
print("Lineas:", len(lista_lineas))
archivo.close()
```

2

```
ruta = r"c:\Programacion2\productos.txt"
archivo = open(ruta,"r")
contenido = archivo.read()
print("Cantidad de caracteres: ", len(contenido))
archivo.close()
```

3

```
ruta = r"c:\Programacion2\productos.txt"
archivo = open(ruta, "r")
lista_diccionarios_productos = list()
for linea in archivo:
    lista = linea.split(";")
    lista_diccionarios_productos.append({"codigo" :
int(lista[0]), "nombre": lista[1], "cantidad":
int(lista[2])})
print(lista_diccionarios_productos)
archivo.close()
```

With open() – Manejo seguro de archivos

```
FILE_PATH = r"c:\Programacion2\datos.txt"
with open(FILE_PATH, “r”) as file:
      for line in file:
            print(line)
```

Forma **recomendada** de abrir archivos en Python. Garantiza que el archivo se cierre automáticamente al terminar el bloque, **incluso si ocurre una excepción**.
“r”: modo de apertura (lectura)
as file: alias para el objeto archivo abierto

if de una sola línea (operador ternario)

```
# [valor_en_caso_True] if [condición] else [valor_en_caso_False]
edad = 20

etapa_edad = "Menor" if edad < 18 else "Adulto"
#  EQUIVALENTE a
if edad < 18:
        etapa_edad = "Menor"
else:
        etapa_edad = "Adulto"
print(etapa_edad)
```