UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

UNIDAD DIDÁCTICA III

DIPLOMATURA EN PYTHON

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo I – Nivel Inicial I.

Unidad III – Tipos de datos II.

Bloques temáticos:

1.- Tupla.

2.- Set.

3.- Formato de salida de datos.

4.- Date.

5.- for/in.

1. Tupla.

Las tuplas son secuencias como las listas pero son INMUTABLES como los strings. Se utilizan para representar colecciones fijas de ítems. Por ejemplo los componentes de un calendario específico.

Se crean con o sin paréntesis, aunque en ocasiones en declaraciones complejas es necesario incluir los paréntesis.

```
tupla1 = 1, 2, 3, 4, 5, 3
print(type(tupla1))
tupla2 = (1, 2, 3, 4, 5, 3)
print(type(tupla2))
```

Retorna:

```
<class 'tuple'>

<class 'tuple'>
```

Las tuplas pueden ser anidadas, y podemos ver que la salida es retornada siempre entre paréntesis.

```
tupla3 = tupla1, tupla2
print(type(tupla3))
print(tupla3)
```

Retorna:

```
<class 'tuple'>

((1, 2, 3, 4, 5, 3), (1, 2, 3, 4, 5, 3))
```

Para determinar un elemento de la tupla lo hacemos de forma análoga a como lo realizamos con las listas, y de la misma forma que con el resto de los tipos de datos contamos con métodos que podemos utilizar mediante notación de punto.

```
tupla4 = tupla3 + (6, 7)
print(tupla4)
# ########################################
# Determinar un elemento
# ########################################
print(tupla4[0])
# ########################################
# Posición de un elemento
# ########################################
print(tupla4.index(6))
# ########################################
# n° de veces que aparece un elemento
# ########################################
print(tupla4.count(7))
```

Retorna:

```
((1, 2, 3, 4, 5, 3), (1, 2, 3, 4, 5, 3), 6, 7)

(1, 2, 3, 4, 5, 3)

2

1
```

Un problema especial es la construcción de tuplas que contienen 0 o 1 elementos: la sintaxis tiene algunas peculiaridades adicionales para acomodarlos. Las tuplas vacías se construyen con un par de paréntesis vacíos; una tupla con un elemento se construye siguiendo un valor con una coma (no es suficiente para encerrar un solo valor entre paréntesis).

```
tupla5 = ()
tupla6 = 'Manzana',
print(len(tupla5))
print(len(tupla6))
print(tupla6)
```

Retorna:

```
0

1
```

```
('Manzana',)
```

2. SET

Es una colección sin orden de objetos únicos e INMUTABLES que soportan operaciones que se corresponden con la teoría matemática de set. Por definición un ítem aparece una sola vez en el set, no importa cuántas veces se agrega. Es como trabajar con los conjuntos que aprendimos en el colegio y de los cuales realizamos operaciones de unión, intersección y diferencia en los “Diagramas de Venn”.

La aplicación de los sets es en casos numéricos y trabajos con bases de datos.

Los sets son como las claves de un diccionario presentadas en forma desordenada. Sus elementos se escriben entre llaves. Vamos ahora a ver un ejemplo:

sets/set1_definicion.py

```
# ########################################
# Crear set
# ########################################
set1 = {1, 2, 3, 4, 5, 3}
print(set1)
print(len(set1))
```

Resultado:

```
{1, 2, 3, 4, 5}
5
```

Nota: Veamos que aunque el set anterior posee 6 elementos, al obtener su longitud con len() el resultado da 5 dado que el set no cuenta los elementos repetidos (en este caso el 3)

Un ejemplo un poco más complejo nos permite ver cómo trabajar algunas operaciones básicas con los sets.

sets/set2_operaciones.py

```
# ########################################
# Crear set
# ########################################
set0 = {1, 2, 3, 4, 5, 3}
set1 = {10, 2, 13, 4, 15, 3, 1}
set2 = {10, 2, 13, 4, 15, 3}
# ########################################
# Crear Intersección
```

```
# ########################################
print(set0 & set1)
# ########################################
# Crear Unión
# ########################################
print(set0 | set1)
# ########################################
# Crear Diferencia
# ########################################
print(set0 - set1)
# ########################################
# Crear Superset
# set1 es un set que contiene todos los
# elementos de set2 y más
# ########################################
print(set1 > set2)
```

Las salidas son respectivamente:

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 10, 13, 15}
{5}
True
```

Set vacío.

Un set vacío debe inicializarse mediante set() ya que {} define un diccionario

Usos de los sets.

Dado que los ítems son guardados solo una vez, pueden usarse para filtrar duplicados, y ya que son desordenados, deberían ordenarse en el proceso. Por ejemplo podría convertirse una lista a set y luego regresar a una lista.

sets/set3_ejemplo1_remover_duplicados.py

```
# ########################################
# Remover duplicados
# ########################################
lista = [1, 2, 1, 3, 2, 4, 5]
mi_set = set(lista)
```

```
print(lista)
print(mi_set)
lista2 = list(mi_set)  # remueve duplicados
print(lista2)
```

La salida nos da:

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

{1, 2, 3, 4, 5}

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

Aplicación

Este tipo de datos nos podría servir por ejemplo para analizar las listas de membresía de un club y realizar estadísticas sobre los deportes o actividades realizadas por los miembros. Supongamos que tenemos las siguientes listas:

```
socios = [“Juan”, “Pedro”, “Susana”, “Anna”, “Sofía”, “Pablo”]
```

Y que en el club se realiza natación y ajedrez:

```
ajedrez = ["Pedro", "Susana", "Anna", "Sofía", "Pablo"]
natacion = ["Juan", "Pedro", "Susana"]

resultado=set(ajedrez).intersection(set(natacion))
print(resultado)
print(type(resultado))
```

Retorna:

```
{'Susana', 'Pedro'}

<class 'set'>
```

También podríamos utilizar los métodos “unión()” o “difference()”, para unir o hallar diferencias de conjuntos.

3. Formato de salida de datos

Hasta ahora hemos utilizado el método print() para imprimir algunos resultados en pantalla, este método posee algunas otras variantes, como es el caso de imprimir varios resultados juntos separados por espacios u otros tipos de caracteres:

```
x = 1
y = 2
z = ['juan']
print(x, y, z, sep=' ')
print(x, y, z, sep=' , ')
```

Retorna:

```
1 2 ['juan']

1 , 2 , ['juan']
```

O agregar alguna instrucción al final de cada línea, como una descripción, tabulación o salto de línea.

```
print(x, y, z, sep=' , ', end='!\n')
```

Retorna:

```
1 , 2 , ['juan']!
```

Secuencia de datos

En ocasiones es útil trabajar con secuencia de datos, supongamos que tenemos la lista

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

Y que asociamos una posición a una referencia de la siguiente manera:

```
a, b, c, d = seq
```

De esta forma al imprimir

```
print(a, b, c, d)
```

Nos retorna:

```
1 2 3 4
```

En el ejemplo anterior también es posible asociar a un grupo de valores una lista , por ejemplo podemos decir que el primer valor está asociado a la letra “a” y el resto a una lista llamada “b” anteponiendo un asterisco a la letra b de la siguiente forma:

```
a, *b = seq
print(a,b, type(b))
```

El resultado que nos retorna es:

```
1 [2, 3, 4] <class 'list'>
```

En el caso de que en lugar de una lista estemos trabajando con un string, b crearía una lista de caracteres.

```
a, *b = 'Manzana'
print(a, b, type(b))
```

Retorna.

```
M ['a', 'n', 'z', 'a', 'n', 'a'] <class 'list'>
```

También contamos con algunas variantes muy útiles, como los formatos literales, veamos cuales son y cómo se utilizan:

Formato literal (f-strings)

Esta es una forma muy útil de darle formato a los datos, debemos comenzar una cadena de texto con “f” o “F” antes de las comillas de apertura o las comillas triples. Dentro de esta cadena podemos escribir una expresión de Python entre los caracteres de apertura y cierre de llaves.

```
nombre = 'Pedro'
edad = 40
print(f'La edad de {nombre} es de {edad} años')
```

Retorna:

```
La edad de Pedro es de 40 años
```

También podríamos utilizar este formato para realizar columnas de dados, agregando luego de lo que estamos imprimiendo, dos puntos seguidos de la mínima cantidad de caracteres que queremos imprimir.

```
nombre = 'Pedro'
edad = 40
print(f'{"fila":30}==>{nombre:10}{edad}')
```

```
fila                          ==>Pedro     40
```

O especificar la cantidad de decimales luego de la coma de un número.

```
valor = 3.141659
print(f'El valor redondeado a 3 dígitos luego de la coma queda: {valor:.3f}.')
```

Retorna:

```
El valor redondeado a 3 dígitos luego de la coma queda: 3.142.
```

O especificar la cantidad de decimales luego de la coma de un número.

Formato con format()

Dar formato con “format()” es bastante similar, también utilizamos las llaves para indicar donde se coloca la variable dentro del string pero podemos hacer referencia por posición utilizando números, veamos un ejemplo:

```
votos = 42572654
voto_en_blanco = 43132495
porcentaje = (votos / (votos + voto_en_blanco)) *100
print('Los votos a favor son: {0} con un porcentaje de: {1:.2f}'.format(votos, porcentaje))
```

Retorna:

```
Los votos a favor son: 42572654 con un porcentaje de: 49.67
```

Notemos que las posiciones se comienzan a numerar desde cero y que en particular el segundo parámetro posee un uno (el cual representa la posición dentro de format()) seguido de dos puntos y a continuación la cantidad de dígitos después de la coma (.2f)

pprint

Existe un módulo que viene en las distribuciones de python y que podemos importar para aumentar la legibilidad llamado pprint, analicemos un ejemplo utilizando diccionarios en donde imprimimos utilizando print() y pprint()

```
import pprint
```

```
juan = {'identificacion': {'nombre': 'Juan', 'apellido':
'Garcia'},
'edad': 24,
'sueldo': 5000,
'profesión': 'Pintor'}
susana = {'identificacion': {'nombre': 'Susana', 'apellido':
'Gomez'},
'edad':25,
'sueldo': 6000,
'profesión': 'Empleada'}
db= {}
db['juan'] = juan
db['susana'] = susana
print(db)
print('------------------------------')

pprint.pprint(db)
```

Retorna:

```
{'juan': {'identificacion': {'nombre': 'Juan', 'apellido': 'Garcia'}, 'edad': 24, 'sueldo': 5000,
'profesión': 'Pintor'}, 'susana': {'identificacion': {'nombre': 'Susana', 'apellido': 'Gomez'},
'edad': 25, 'sueldo': 6000, 'profesión': 'Empleada'}}

------------------------------

{'juan': {'edad': 24,

          'identificacion': {'apellido': 'Garcia', 'nombre': 'Juan'},

          'profesión': 'Pintor',

          'sueldo': 5000},

'susana': {'edad': 25,

            'identificacion': {'apellido': 'Gomez', 'nombre': 'Susana'},

            'profesión': 'Empleada',

            'sueldo': 6000}}
```

4. Date

Trabajar con fechas es algo muy importante y que realizamos muy a menudo, existen dos librerías muy útiles:

```
import datetime
import calendar
```

Con datetime podemos obtener datos de la fecha actual, desde el año a microsegundos

```
import datetime
import calendar

datetime.datetime.now()
print(datetime.datetime.utcnow())
print(datetime.datetime.now())
print(datetime.datetime.now().day)
print(datetime.datetime.now().month)
print(datetime.datetime.now().minute)
print(datetime.datetime.now().second)
print(datetime.datetime.now().microsecond)
```

Contamos con una serie de abreviaciones que nos permiten especificar claramente el formato de fecha e incluso realizar operaciones entre fechas:

```
print(datetime.datetime.today().strftime("%H:%M:%S--%d/%m/%y"))
"""
%a - Nombre del día de la semana
%A - Nombre del día completo
%b - Nombre abreviado del mes
%B - Nombre completo del mes
%c - Fecha y hora actual
%d - Día del mes
%H - Hora (formato 24 horas)
%I - Hora (formato 12 horas)
%j - Día del año
%m - Mes en número
%M- Minutos
%p - Equivalente de AM o PM
%S - Segundos
```

```
%U - Semana del año (domingo como primer día de la semana)
%w - Día de la semana
%W - Semana del año (lunes como primer día de la semana)
%x - Fecha actual
%X - Hora actual
%y - Número de año (14)
%Y - Número de año entero (2014)
%Z - Zona horaria
"""
#Diferencia de fechas 1
actual = datetime.datetime.now()
anterior = datetime.datetime(1975, 9, 15, 0, 0, 0)
print(actual-anterior)

#Diferencia de fechas 2
hoy = datetime.date.today()
hace5 = datetime.timedelta(days=5)
print(hoy)
print(hace5)
print(hoy - hace5) #Resto 5 días
```

for

La estructura for, es diferente a otros lenguajes de programación, por lo que más adelante daremos un análisis completo de esta estructura. Por ahora y para comenzar a realizar ejercicios simples vamos a definir su forma básica como sigue:

```
for objetivo in objeto:
    código1
```

Esta forma poco típica de utilización de un bucle for puede sorprender a quien viene de otros lenguajes de programación, sin embargo es de uso frecuente en Python y nos permite ejecutar un código dependiendo de si un determinado objeto se encuentra dentro de otro o no.