1000 PROGRAMADORES

Introducción a la
Programación en Python

MÓDULO 2

Variables.
Números, enteros y flotantes. Cadenas de Texto.
Listas. Tuplas. Diccionarios. Conjuntos.
Índices y slicing. Métodos de colecciones.
Operadores lógicos. Operadores relacionales y de asignación.

Ministerio de Educación
Cultura, Ciencia y Tecnología
Gobierno de Salta

Ministerio de Economía
y Servicios Públicos
Gobierno de Salta

Universidad
Nacional de Salta

INTRODUCCIÓN

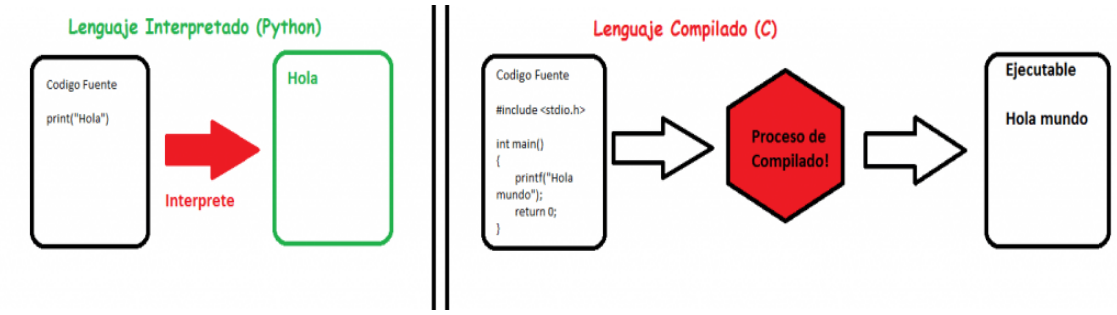
Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Y define este como un lenguaje multiparadigma, debido a que soporta orientación a objetos, programación imperativa y en menor medida programación funcional. Es interpretado de tipado dinámico y multiplataforma.

Vamos a tratar de explicar cada característica del lenguaje de la manera más simple posible.

- Interpretado

Normalmente los lenguajes de programación se pueden agrupar en Interpretados y Compilados según la forma en la que son traducidos. Los lenguajes se inventaron para facilitar al programador el desarrollo aplicaciones, imagina que nuestro computador trabaja en código binario y sería muy difícil programar todo en unos y ceros, además del tiempo que eso llevaría.

Por eso al escribir un código en realidad lo que estamos haciendo es hablar un lenguaje más fácil de comprender para nosotros y que luego será traducido a lenguaje de máquina que es lo que puede entender el procesador. Entonces, cómo se realiza esta "traducción" es lo que los agrupa en Interpretados o Compilados:



Los lenguajes Compilados son aquellos en los que el código del programador es traducido por completo de una sola vez mediante un proceso llamado "Compilado" para ser ejecutado por un sistema predeterminado. Entre los más comunes encontramos "C", "C++", Java, etc.

Y los lenguajes Interpretados son aquellos en los que el código del programador es traducido mediante un intérprete a medida que es necesario. Entre los más comunes encontramos "Python", "Ruby", "Javascript", etc.

El que Python sea interpretado nos presenta ventajas:

- No necesitamos compilar ahorrándonos mucho tiempo en el desarrollo y prueba de una aplicación.
- Nuestro código fuente puede ser ejecutado en cualquier software siempre y cuando este disponga del intérprete (Windows, Linux, Mac, Android, Web).

- Multiparadigma

Esto nos dice que Python es un lenguaje que soporta más de un paradigma, suponiendo paradigma como modelo de desarrollo (y cada lenguaje tiene el suyo). El paradigma más sobresaliente en la programación en Python es el **Orientado a Objetos.**

La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones entre los objetos. Python también permite la programación imperativa, programación funcional y programación orientada a aspectos.

- De Tipado Dinámico

Es de tipado dinámico cuando una variable puede tomar **diferentes valores** de **distintos tipos** en **diferentes momentos.** En Python las variables son declaradas por su contenido y no por su contenedor, lo que nos va a permitir cambiar el valor y tipo de una variable durante la ejecución sin necesidad de volver a declarar..

Por ejemplo supongamos que X es una variable, y en este caso X = 1. Si bien X está almacenando un número, igualmente podemos almacenar una letra durante la ejecución del programa en la misma variable X mediante una instrucción, por ejemplo X = “a”.

Veamos una variable como un bloque de memoria donde podemos “almacenar” un dato. O como una cajita donde guardar cosas..

En python:

- Si dentro de la variable colocamos números, la variable será de tipo numérica.
- Ahora, si colocamos letras, la variable será de tipo texto.
- Y si en ella colocamos un booleano (Verdadero o Falso), será de tipo booleano.

Es decir, vamos a guardar en ella lo que queremos y python automáticamente detectara su tipo, no hace falta que se lo indiquemos.

Resumiendo

Python es un lenguaje que todo el mundo debería conocer. Su sintaxis simple, clara y sencilla; el tipado dinámico, el gestor de memoria, la gran cantidad de librerías disponibles y la potencia del lenguaje, entre otros, hacen que desarrollar una aplicación en Python sea sencillo, muy rápido y, lo que es más importante, divertido.

La sintaxis de Python es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo. Por este motivo se trata además de uno de los mejores lenguajes para comenzar a programar.

Python no es adecuado sin embargo para la programación de bajo nivel o para aplicaciones en las que el rendimiento sea crítico.

Algunos casos de éxito en el uso de Python son Google, Yahoo, la NASA, Industrias Light & Magic, y todas las distribuciones Linux, en las que Python cada vez representa un tanto por ciento mayor de los programas disponibles.

VARIABLES

Las variables son uno de los componentes básicos de cualquier programa. En su esencia, un programa está compuesto por datos e instrucciones que manipulan esos datos. Normalmente, los datos se almacenan en memoria (RAM) para que podamos acceder a ellos.

Una variable es una forma de identificar, de forma sencilla, un dato que se encuentra almacenado en la memoria del ordenador. Es un contenedor en el que se almacena un dato, el cuál, puede cambiar durante el flujo del programa. Una variable nos permite acceder fácilmente a dicho dato para ser manipulado y transformado.

Por ejemplo, si queremos mostrar el resultado de sumar 1 + 2 debemos indicarle al programa dónde se encuentra dicho dato en memoria y, para ello, hacemos uso de una variable:

```
#Guardamos en la variable suma el resultado de 1 + 2
suma = 1 + 2
# Accedemos al resultado de 1 + 2 a través de la variable suma
print(suma)
```

```
3
```

---

salida

Como vimos en el ejemplo anterior, para asignar un valor (un dato) a una variable se utiliza el operador de asignación =.

En la operación de asignación se ven involucradas tres partes:

- El operador de asignación =
- Un identificador o nombre de variable, a la izquierda del operador
- Un literal, una expresión, una llamada a una función o una combinación de todos ellos a la derecha del operador de asignación

```
# Asigna a la variable <a> el valor 1
a = 1
# Asigna a la variable <a> el resultado de la expresión 3 * 4
a = 3 * 4
# Asigna a la variable <a> la cadena de caracteres 'Python'
a = 'Python'
```

Cuando asignamos un valor a una variable por primera vez, se dice que en ese lugar se *define* e *inicializa* la variable. En un script o programa escrito en Python, podemos definir variables en cualquier lugar del mismo. Sin embargo, es una buena práctica definir las variables que vayamos a utilizar al principio.

Si intentamos usar una variable que no ha sido definida/inicializada previamente, el intérprete nos mostrará un error:

```
print(a)
```

```
_____________________________________________________________________________
NameError                                 Traceback (most recent call last)
<ipython-input-8-bca0e2660b9f> in <module>()
----> 1 print(a)
NameError: name 'a' is not defined
_____________________________________________________________________________
```

salida

Para modificar el valor de una variable en Python, basta con asignarle un nuevo valor en cualquier momento y lugar después de la definición.

```
a = 1              # Asigna a la variable <a> el valor 1
print(a)
a = 3              # Modifica el calor de la variable <a> de 1 a 3
print(a)
```

```
_____________________________________________________________________________
1
3
_____________________________________________________________________________
```

salida

TIPOS DE DATOS BÁSICOS

En Python los tipos básicos se dividen en:

- Números, como pueden ser 3 (entero) , 15.57 (de coma flotante) o 7 + 5j (complejos)
- Cadenas de texto, como “Hola Mundo”
- Valores booleanos: True (cierto) y False (falso).

Vamos a crear un par de variables a modo de ejemplo. Una de tipo cadena y una de tipo entero:

```
# esto es una cadena
c = "Hola Mundo"
# y esto es un entero
e = 23
# podemos comprobarlo con la función type
type(c),type(e)
```

```
(str, int)
```

salida

En Python, a diferencia de muchos otros lenguajes, no se declara el tipo de la variable al crearla. En Java, por ejemplo, escribiríamos:

```
String c = “Hola Mundo”;
int e = 23;
```

Este ejemplo también sirve para presentar los comentarios inline en Python: cadenas de texto que comienzan con el carácter # y se ignoran totalmente. Hay más tipos de comentarios, que veremos más adelante.

Números

Como decíamos, en Python se pueden representar números enteros, reales y complejos.

Enteros

El tipo de los números enteros es ***`int`***. Este tipo de dato comprende el conjunto de

todos los números enteros, pero como dicho conjunto es infinito, en Python el conjunto está limitado realmente por la capacidad de la memoria disponible. No hay un límite de representación impuesto por el lenguaje.

Un número de tipo int se crea a partir de un literal que represente un número entero o bien como resultado de una expresión o una llamada a una función.

Ejemplos:

```
a = -1      # a es de tipo int y su valor es -1
b = a + 2   # b es de tipo int y su valor es 1
print(b)
```

```
1
```

salida

También podemos representar los números enteros en formato *binario*, *octal o hexadecimal*.

Los números octales se crean anteponiendo el valor `0o` a una secuencia de dígitos octales (del 0 al 7). Para crear un número entero en hexadecimal, hay que anteponer `0x` a una secuencia de dígitos en hexadecimal (del 0 al 9 y de la A la F). En cuanto a los números en binario, se antepone `0b` a una secuencia de dígitos en binario (0 y 1).

```
diez = 10
diez_binario = 0b1010
diez_octal = 0o12
diez_hex = 0xa
print(diez)
print(diez_binario)
print(diez_octal)
print(diez_hex)
```

```
10
10
10
10
```

salida

Números de punto flotante

Los números de punto flotante representan al conjunto de los números reales. Hay que tener en cuenta algunos detalles. Veamos un ejemplo:

```
1.1 + 2.2
```

```
3.3000000000000003
```

salida

Al igual que ocurre con los números enteros, los números reales son infinitos y, por tanto, es imposible representar todo el conjunto de números reales con un ordenador.

Para representar el mayor número posible de los números reales con las limitaciones de memoria (tamaños de palabra de 32 y 64 bits), se adaptó la notación científica de representación de números reales al sistema binario (que es el sistema que se utiliza en programación para representar los datos e instrucciones).

El caso es que la suma de la representación en punto flotante en binario del número 1,1 y de la representación en punto flotante en binario del número 2,2, dan como resultado 3,3000000000000003

Pero hay más casos, como por ejemplo la representación del número 1/3. En algún momento, el ordenador tiene que truncar el número periódico resultante.

La explicación final es que los números de punto flotante se representan en el hardware del ordenador como fracciones de base 2 (binarias). Y el problema está en que la mayoría de las fracciones decimales no se pueden representar de forma exacta como fracciones binarias porque tienen infinitos números decimales. Una consecuencia es que, en general, los números decimales de punto flotante que usamos en nuestras aplicaciones son una aproximación de los números binarios de punto flotante realmente almacenados en la máquina.

Este tipo de datos en Python es float. Que puede ser usado para representar cualquier número real (siempre teniendo en cuenta que es una aproximación lo más precisa posible). Por tanto para longitudes, pesos, frecuencias, etc, en los que prácticamente es lo mismo 3,3 que 3,3000000000000003. Además es posible darle formato al número de la siguiente manera:

```
real = 1.1 + 2.2       # real es un float
print(real)
print(f'{real:.2f}')
```

```
3.3000000000000003     # representación aproximada de 3.3
3.30                   #   real   mostrando   únicamente   2   cifras
decimales
```

---

salida

Números complejos

El último tipo de dato numérico básico que tiene Python es el de los números complejos, complex.

Los números complejos tienen una parte real y otra imaginaria y cada una de ellas se representa como un float. Para crear un número complejo, se sigue la siguiente estructura `<parte_real>+<parte_imaginaria>j`. Y se puede acceder a la parte real e imaginaria a través de los atributos **`real`** e **`imag`**:

```
complejo = 1+2j
complejo.real
```

```
1.0
```

---

salida

```
complejo = 1+2j
complejo.imag
```

```
2.0
```

---

salida

Aritmética de los tipos numéricos

Con todos los tipos numéricos se pueden aplicar las operaciones de la aritmética: suma, resta, producto, división.

En Python está permitido realizar una operación aritmética con números de distinto tipo. En este caso, el tipo numérico «más pequeño» se convierte al del tipo «más grande», de manera que el tipo del resultado siempre es el del tipo *mayor*. Entendemos que el tipo int es menor que el tipo float que a su vez es menor que el tipo complex.

Por tanto, es posible, por ejemplo, sumar un int y un float:

```
res_float=1 + 2.0
res_complejo=2+3j + 5.7
print(res_float)
print(res_complejo)
```

```
3.0                 # el resultado de (int + float) es float
(7.7+3j)            # el resultado de (complex + float) es complex
```

salida

Tipo booleano

En Python la clase que representa los valores booleanos es ***bool***. Esta clase solo se puede instanciar con dos valores/objetos: True para representar verdadero y False para representar falso.

Una particularidad del lenguaje es que cualquier objeto puede ser usado en un contexto donde se requiera comprobar si algo es *verdadero* o *falso*. Por tanto, cualquier objeto se puede usar en la condición de un if o un while (son estructuras de control que veremos en tutoriales posteriores) o como operando de una operación booleana.

Por defecto, cualquier objeto es considerado como verdadero con dos excepciones:

- Que implemente el método __bool__() y este devuelva False.
- Que implemente el método __len__() y este devuelva 0.

Además, los siguientes objetos/instancias también son consideradas falsas:

- None
- False
- El valor cero de cualquier tipo numérico: 0, 0.0, 0j, …
- Secuencias y colecciones vacías : '', (), [], {}, set(), range(0)

Más adelante desarrollaremos el significado de métodos y de otros tipos de datos.

Tipo cadena de caracteres

Otro tipo básico de Python, e imprescindible, son las secuencias o cadenas de caracteres. Este tipo es conocido como *string* aunque su clase verdadera es `str`. Formalmente, un *string* es una secuencia inmutable de caracteres en formato Unicode.

Para crear un string, simplemente hay que encerrar entre comillas simples ' o dobles "" una secuencia de caracteres.

Se puede usar indistintamente comillas simples o dobles, con una particularidad. Si en la cadena de caracteres se necesita usar una comilla simple, tienes dos opciones: usar comillas dobles para encerrar el string, o bien, usar comillas simples pero anteponer el carácter \ a la comilla simple del interior de la cadena. El caso contrario es similar.

Veamos todo esto con un ejemplo:

```
hola_1 = "Hola Mundo"
hola_2 = 'Hola Mundo'
hola_3 = 'Hola "Mundo"'
hola_4 = 'Hola \'Mundo\''
hola_5 = "Hola 'Mundo'"
print(hola_1,'-', hola_2)
print(hola_3)
print(hola_4, '-',hola_5)
```

```
Hola Mundo - Hola Mundo
Hola "Mundo"
Hola 'Mundo' - Hola 'Mundo'
```

---

salida

A diferencia de otros lenguajes, en Python no existe el tipo «carácter». No obstante, se puede simular con un string de un solo carácter:

```
caracter_a = 'a'
print(caracter_a)
```

```
a
```

---

salida

COLECCIONES DE DATOS

Además de los tipos básicos, otros tipos fundamentales de Python son los tipos compuestos y se utilizan para agrupar juntos varios valores. Entre ellos tenemos:

- Listas: son secuencias mutables de valores.
- Tuplas: son secuencias inmutables de valores.
- Conjuntos: se utilizan para representar conjuntos donde no pueden existir dos objetos iguales dentro de él.
- Diccionarios: son tipos especiales de contenedores en los que se puede acceder a sus elementos a partir de una clave única.

Listas

Las listas en Python son un tipo contenedor, compuesto, que se usan para almacenar conjuntos de elementos relacionados del mismo tipo o de tipos distintos.Tienen la particularidad de ser mutables, es decir, que su contenido se puede modificar después de haber sido creada.

Para crear una lista en Python, simplemente hay que encerrar una secuencia de elementos separados por comas entre corchetes [].

Ejemplos de creación de listas:

```
# elementos del mismo tipo (enteros)
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# elementos de distinto tipo (enteros, float, string)
elementos = [3, 'a', 8, 7.2, 'hola']
# elementos compuestos de distinto tipo (listas dentro de listas)
lista = [1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola']
```

Las listas también se pueden crear usando el constructor de la clase, `list(iterable)`. En este caso, el constructor crea una lista cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto *iterable* puede ser o una secuencia, un contenedor que soporte la iteración o un objeto iterador (más adelante explicaremos los conceptos de clase y de objeto iterador).

Por ejemplo, el tipo *str* también es un tipo secuencia. Si pasamos un string al constructor list() creará una lista cuyos elementos son cada uno de los caracteres de la cadena:

```
vocales = list('aeiou')
vocales
```

```
['a', 'e', 'i', 'o', 'u']
```

salida

También podemos crear listas sin especificar los elementos que contienen, porque al ser mutables los podemos agregar después. A continuación se muestran dos alternativas de crear una lista vacía:

```
lista_1 = []
print(lista_1)
lista_2 = list()
print(lista_2)
```

```
[]
[]
```

salida

Índices y Slicing

Acceso a los elementos de una lista en Python

Para acceder a un elemento de una lista se utilizan los índices. Un índice es un número entero que indica la posición de un elemento en una lista. El primer elemento se encuentra en el índice 0. Por ejemplo, en una lista con 4 elementos, los índices de cada uno de los ítems serían 0, 1, 2 y 3.

```
lista = ['a', 'b', 'd', 'i', 'j']
lista[0]  # Primer elemento de la lista. Índice 0
lista[3]  # Cuarto elemento de la lista. Índice 3
```

```
a
i
```

salida

Si se intenta acceder a un índice que está fuera del rango de la lista, el intérprete lanzará la excepción `IndexError`.

```
lista = [1, 2, 3]  # Los índices válidos son 0, 1 y 2
lista[8]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-2-3cba9a049cf3> in <module>()
      1 lista = [1, 2, 3]  # Los índices válidos son 0, 1 y 2
----> 2 lista[8]
IndexError: list index out of range
```

salida

Si se utiliza un índice que no es un número entero, se lanzará la excepción `TypeError`:

```
lista = [1, 2, 3]  # Los índices válidos son 0, 1 y 2
lista[1.0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-e83608370883> in <module>()
      1 lista = [1, 2, 3]  # Los índices válidos son 0, 1 y 2
----> 2 lista[1.0]
TypeError: list indices must be integers or slices, not float
```

salida

Como hemos visto, las listas pueden contener otros elementos de tipo secuencia de forma anidada. Por ejemplo, una lista donde uno de sus ítems es otra lista. Del mismo modo, se puede acceder a los elementos de estos tipos usando índices compuestos o anidados:

```
lista = ['a', ['d', 'b'], 'z']
lista[1][1]
# lista[1] hace referencia a la posición de la lista anidada
# lista[1][1] hace referencia al 2do elemento de la lista anidada
```

```
b
```

salida

Acceso a un subconjunto de elementos

También es posible acceder a un subconjunto de elementos de una lista utilizando rangos en los índices. Esto es usando el operador [:]:

```
vocales = ['a', 'e', 'i', 'o', 'u']
vocales[2:3]  # Elementos desde el índice 2 hasta el índice 3-1
```

```
['i']
```

salida

```
vocales = ['a', 'e', 'i', 'o', 'u']
vocales[2:4]  # Elementos desde el 2 hasta el índice 4-1
```

```
['i', 'o']
```

salida

```
vocales = ['a', 'e', 'i', 'o', 'u']
vocales[1:]  # Elementos desde el índice 1
```

```
['e', 'i', 'o', 'u']
```

salida

```
vocales = ['a', 'e', 'i', 'o', 'u']
vocales[:3]  # Elementos hasta el índice 3-1
```

```
['a', 'e', 'i']
```

salida

Otras tareas que pueden realizarse sobre una lista son:

- Recorrer una lista
- Añadir/Modificar/Eliminar elementos
- Determinar la longitud de la lista
- Determinar si un elemento pertenece a una lista

Estas y otras tareas serán estudiadas en el Módulo de Funciones de Python.

Tuplas

La clase `tuple` en Python es un tipo contenedor, compuesto, que en un principio se pensó para almacenar grupos de elementos heterogéneos, aunque también puede contener elementos homogéneos. Tiene la particularidad de ser inmutables, es decir, que su contenido NO se puede modificar después de haber sido creada.

En general, para crear una tupla en Python simplemente hay que definir una secuencia de elementos separados por comas. Por ejemplo:

```
numeros = 1, 2, 3, 4, 5                    # Elementos homogéneos
print (numeros)
elementos = 3, 'a', 8, 7.2, 'hola'         # Elementos heterogéneos
print(elementos)
tup = 1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola'
print(tup)
```

```
(1, 2, 3, 4, 5)
(3, 'a', 8, 7.2, 'hola')
(1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola')
```

---

salida

Otras formas de crear una tupla en Python:

- Tupla vacía: usa paréntesis () o el constructor tuple() sin parámetros.
- Tupla con un único elemento: elem, o (elem,). Siempre se añade una coma.
- Usando el constructor tuple(iterable), donde el constructor crea una tupla cuyos elementos son los mismos y en el mismo orden que los del iterable.

```
tupla1=()
tupla2='a',
tupla3 = tuple('aeiou')
print(tupla1)
print(tupla2)
print(tupla3)
```

```
()
('a',)
('a', 'e', 'i', 'o', 'u')
```

---

salida

Índices y Slicing

Acceso a los elementos de una tupla en Python

Para acceder a un elemento de una tupla se utilizan los índices. Un índice es un número entero que indica la posición de un elemento en una tupla. El primer elemento de una tupla siempre comienza en el índice 0.

Por ejemplo, en una tupla con 3 elementos, los índices de cada uno de los ítems serían 0, 1 y 2.

```
tupla = ('a', 'b', 'd')
tupla[0]  # Primer elemento de la tupla. Índice 0
```

```
'a'
```

salida

Al igual que con Listas, si se intenta acceder a un índice que está fuera del rango de la tupla, el intérprete lanzará la excepción IndexError. Sii se utiliza un índice que no es un número entero, se lanzará la excepción TypeError:

También es posible acceder a un subconjunto de elementos de una tupla utilizando el operador [:].

Los ejemplos de estas tareas son idénticos a los mostrados en la sección de listas, usando `tuple` en vez de `list`.

Modificar una tupla

Como ya se indicó, las tuplas son objetos inmutables. No obstante, las tuplas pueden contener objetos u otros elementos de tipo secuencia, por ejemplo, una lista. Estos objetos, si son mutables, sí se pueden modificar:

```
tupla = (1, ['a', 'b'], 'hola', 8.2)
tupla[1].append('c')        # tupla[1] hace referencia a la lista
tupla
```

```
(1, ['a', 'b', 'c'], 'hola', 8.2)
```

salida

Otras tareas que pueden realizarse sobre una tupla son:

- Recorrer la tupla
- Determinar la longitud de la tupla
- Determinar si un elemento pertenece a una tupla

Estas y otras tareas serán estudiadas en el Módulo de Funciones de Python.

Recuerda que las tuplas son un tipo secuencial que no se puede modificar. Se usan cuando no quieres que los elementos de una lista se deban modificar o para almacenar las claves de un diccionario.

Diccionarios

La clase dict de Python es un tipo mapa que asocia claves a valores. A diferencia de los tipos secuenciales como list, tuple o str, que son indexados por un índice numérico, los diccionarios son indexados por claves. Estas claves siempre deben ser de un tipo inmutable, concretamente un tipo hashable (un objeto es hashable si tiene un valor de hash que no cambia durante todo su ciclo de vida).

Piensa en un diccionario como un contenedor de pares ***clave:valor***, en el que la clave puede ser de cualquier tipo hashable y es única en el diccionario que la contiene. Generalmente, se suelen usar como claves los tipos *int* y *str*.

Las principales operaciones con diccionarios son:

- Almacenar un valor asociado a una clave
- Recuperar un valor a partir de una clave.

Esta es la esencia de los diccionarios y es aquí donde son realmente importantes. En un diccionario, el acceso a un elemento a partir de una clave es una operación rápida, eficaz y que consume pocos recursos si se compara con la forma en que se hace con otros tipos de datos.

Otras características a resaltar de los diccionarios:

- Es un tipo mutable, es decir, su contenido se puede modificar después de haber sido creado.
- Es un tipo ordenado. Preserva el orden en que se insertan los pares *clave: valor*.

Creación de Diccionarios

La más simple es encerrar una secuencia de pares *clave: valor* separados por comas entre llaves {}

```
d = {1: 'hola', 89: 'Pythonista', 'a': 'b', 'c': 27}
d
```

```
{1: 'hola', 89: 'Pythonista', 'a': 'b', 'c': 27}
```

---

salida

En el diccionario anterior, los enteros 1 y 89 y las cadenas 'a' y 'c' son las claves. Es posible mezclar claves y valores de distinto tipo sin problema.

Para crear un diccionario vacío, simplemente asigna a una variable el valor {}.

```
d = {}            # Diccionario vacío
print(d)
```

```
{}
```

---

salida

También se puede usar el constructor de la clase dict() de varias maneras:

- **Sin parámetros**. Esto creará un diccionario vacío.
- Con pares **clave: valor** encerrados entre llaves.
- **Con argumentos con nombre**. El nombre del argumento será la clave en el diccionario. En este caso, las claves solo pueden ser identificadores válidos y mantienen el orden en el que se indican. No se podría, por ejemplo, tener números enteros como claves.
- **Pasando un iterable**. En este caso, cada elemento del iterable debe ser también un iterable con solo dos elementos. El primero se toma como clave del diccionario y el segundo como valor. Si la clave aparece varias veces, el valor que prevalece es el último.

```
#Argumentos con nombre
d2 = dict(uno=1, dos=2, tres=3)
print(d2)
```

```
{'uno': 1, 'dos': 2, 'tres': 3}
```

---

salida

```
d1 = {'uno': 1, 'dos': 2, 'tres': 3}
print(d1)   # Pares clave: valor encerrados entre llaves
d2 = dict({'uno': 1, 'dos': 2, 'tres': 3})
print(d2)   # Pares clave: valor encerrados entre llaves
d3 = dict([('uno', 1), ('dos', 2), ('tres', 3)])
print(d3)   # Iterable que contiene iterables con dos elementos
d4 = dict()
print(d4)   # Diccionario vacío usando el constructor
```

```
{'uno': 1, 'dos': 2, 'tres': 3}
{'uno': 1, 'dos': 2, 'tres': 3}
{'uno': 1, 'dos': 2, 'tres': 3}
{}
```

salida

Acceso a los elementos de un Diccionario

Acceder a un elemento de un diccionario es una de las principales operaciones por las que existe este tipo de dato. El acceso a un valor se realiza mediante indexación de la clave. Para ello, simplemente encierra entre corchetes la clave del elemento d[clave]. En caso de que la clave no exista, se lanzará la excepción KeyError.

```
d = {'uno': 1, 'dos': 2, 'tres': 3}
print(d['dos'])
```

```
2
```

salida

```
d = {'uno': 1, 'dos': 2, 'tres': 3}
print(d[4])
```

```
KeyError                                  Traceback (most recent call last)
<ipython-input-2-0dec6836298a> in <module>()
      1 d = {'uno': 1, 'dos': 2, 'tres': 3}
      2 print(d['dos'])
----> 3 print(d[4])
KeyError: 4
```

salida

La clase dict también ofrece el método get(clave[, valor por defecto]). Este método devuelve el valor correspondiente a la clave clave. En caso de que la clave no exista no lanza ningún error, sino que devuelve el segundo argumento valor por defecto. Si no se proporciona este argumento, se devuelve el valor None.

```
# Devuelve 1 como valor dado que corresponde a la clave dada
d = {'uno': 1, 'dos': 2, 'tres': 3}
print(d.get('uno'))
# Devuelve 4 como valor por defecto si no encuentra la clave
d.get('cuatro', 4)
print(d.get('cuatro', 4))
# Devuelve None como valor por defecto si no encuentra la clave
a = d.get('cuatro')
print(a)
```

```
1
4
None
```

salida

Otras tareas que pueden realizarse sobre un diccionario son:

- Recorrer el diccionario
- Añadir/Modificar/Eliminar un elemento
- Determinar el número de elementos
- Comprobar si un elemento está en un diccionario
- Comparar/Anidar diccionarios
- Obtener listado de clave de un diccionario

Conjuntos

El tipo set en Python es utilizado para trabajar con conjuntos de elementos. La principal característica de este tipo de datos es que es una colección cuyos elementos no guardan ningún orden y que además son únicos.

Estas características hacen que los principales usos de esta clase sean conocer si un elemento pertenece o no a una colección y eliminar duplicados de un tipo secuencial (list, tuple o str).

Además, esta clase también implementa las típicas operaciones matemáticas sobre conjuntos: unión, intersección, diferencia, etc.

Para crear un conjunto, basta con encerrar una serie de elementos entre llaves {}, o bien usar el constructor de la clase `set`() y pasarle como argumento un objeto iterable (como una lista, una tupla, una cadena).

```
# Crea un conjunto con una serie de elementos entre llaves
# Los elementos repetidos se eliminan
c = {1, 3, 2, 9, 3, 1}
c
```

```
{1, 2, 3, 9}
```

salida

```
# Crea un conjunto a partir de un string
# Los caracteres repetidos se eliminan
a = set('Armando un conjunto')
print(a)
# Crea un conjunto a partir de una lista
# Los elementos repetidos de la lista se eliminan
unicos = set([3, 5, 6, 1, 5])
print(unicos)
```

```
{'r', 'u', 'o', 'A', 'm', ' ', 'j', 't', 'd', 'n', 'a', 'c'}
{1, 3, 5, 6}
```

salida

Para crear un conjunto vacío, simplemente llama al constructor set() sin parámetros.

```
s = set()
print(s)
```

```
set()
```

salida

Acceso a los elementos de un conjunto

Dado que los conjuntos son colecciones desordenadas, en ellos no se guarda la posición en la que son insertados los elementos como ocurre en los tipos list o tuple.

Es por ello que no se puede acceder a los elementos a través de un índice. Sin embargo, sí se puede acceder y/o recorrer todos los elementos de un conjunto usando un bucle for:

```
mi_conjunto = {1, 3, 2, 9, 3, 1}
for e in mi_conjunto:
  print(e)
```

```
1
2
3
9
```

salida

Más adelante estudiaremos la sentencia ***for***, características y aplicaciones.

Además de las tareas mencionadas para otros tipos de datos:

- Añadir/Eliminar un elemento
- Determinar el número de elementos
- Comprobar si un elemento está en un conjunto

En particular podemos agregar las operaciones del álgebra de conjuntos:

- Unión
- Intersección
- Diferencia
- Diferencia Simétrica
- Inclusión de conjuntos
- Conjuntos disjuntos
- Igualdad

OPERADORES

Los operadores son símbolos reservados por el lenguaje que se utilizan para llevar a cabo operaciones sobre uno o más operandos. Los operandos pueden ser variables, literales, el valor devuelto por una expresión o el valor devuelto por una función.

Operadores Aritméticos

Los operadores aritméticos permiten realizar las diferentes operaciones aritméticas del álgebra. Estos operadores Python son de los más utilizados:

Operador	Descripción	Ejemplos	Resultado
+	Sumar	2 + 2	4
-	Restar	3 – 2	1
*	Multiplicar	2 * 2	4
/	Dividir	4 / 3	1.333333
%	Módulo: Devolverá el resto de la división	4 % 2	0
**	Exponente: Exponencial de un número	3**2	9
//	División devuelve el entero de la misma	4//3	1

Operadores de Pertenencia

Los operadores de pertenencia se utilizan para comprobar si un valor o variable se encuentran en una secuencia (list, tuple, dict, set o str).

Operador	Descripción	Ejemplos	Resultado
In	El operador **In** devuelve True si un elemento se encuentra dentro de otro.	a = [3, 4] 3 in a	True Porque 3 se encuentra en “a”
Not in	El operador **Not In** devuelve True si un elemento no se encuentra dentro de otro.	a = [3, 4] 5 in a	True Porque 5 no se encuentra en “a”
Is	El operador **Is** devuelve True si los elementos son exactamente iguales.	x = 10 y = 10 x is y	True Porque ambas variables tienen el mismo valor, son iguales.
Not Is	El operador **not is** devuelve True si los elementos no son exactamente iguales.	x = 10 y = 111 x not is	True Porque las variables no tienen el mismo valor, son diferentes.

Operadores de Asignación

El operador de asignación se utiliza para asignar un valor a una variable. Existe el operador simple "=" y otros de asignación compuestos que realizan una operación básica sobre la variable a la que se le asigna el valor.

Operador	Descripción	Ejemplos	Resultado
=	Asigna un valor a un elemento. Puede ser variable, lista, diccionario, tupla, etc.	a = 2 + 2	"a" vale 4
+=	El primer elemento es igual a la **suma** del primer elemento con el segundo. Se suele utilizar como contador.	b += 1	b = b + 1 Cada vez que se ejecute esta instrucción se le sumara 1 a "b"
-=	El primer elemento es igual a la **resta** del primer elemento con el segundo. Se suele utilizar como contador negativo.	b -= 1	b = b – 1 Cada vez que se ejecute esta instrucción se le restara 1 a "b"
*=	El primer elemento es igual a la **multiplicación** del primer elemento con el segundo.	b *= 2	b = b * 2 Cada vez que se ejecute esta instrucción "b" se multiplicará por dos y se le asignará el valor del resultado.
/=	El primer elemento es igual a la **división** del primer elemento con el segundo.	b /= 2	b = b / 2 Cada vez que se ejecute esta instrucción "b" se dividirá por dos y se le asignará el valor del resultado.
%=	El primer elemento es igual al **resto de la división** del primer elemento con el segundo.	b %= 2	b = b % 2 Cada vez que se ejecute esta instrucción "b" se dividirá por dos y se le asignará el valor del resultado
**=	El primer elemento es igual al **resultado del exponente** del primer elemento con el segundo.	b **= 2	b = b ** 2 Cada vez que se ejecute esta instrucción "b" se expondrá por dos y se le asignará el valor del resultado

Operadores de Comparación

Los operadores de comparación se utilizan, como su nombre indica, para comparar dos o más valores. El resultado de estos operadores siempre es True o False.

Operador	Descripción	Ejemplos	Resultado
==	Si dos valores son **exactamente** iguales devuelve True	2 == 2 2 == 3	True False
!=	Si dos valores son diferentes devuelve True	2 != 5 2 != 2	True False
>	Si el valor de la izquierda **es mayor que el de la derecha** devuelve True	4 > 2 1 > 2	True False
<	Si el valor de la izquierda **es menor que el de la derecha** devuelve True	1 < 2 4 < 2	True False
>=	Si el valor de la izquierda **es mayor o igual que el de la derecha** devuelve True	4 >= 2 1 >= 2 4 >= 4	True False True
<=	Si el valor de la izquierda **es menor o igual que el de la derecha** devuelve True	4 <= 6 1 <= 0 4 <= 4	True False

Operador de concatenación de cadenas de caracteres

Una de las operaciones más básicas cuando se trabaja con cadenas de caracteres es la concatenación. Esto consiste en unir dos cadenas en una sola, siendo el resultado un nuevo string.

La forma más simple de concatenar dos cadenas en Python es utilizando el operador de concatenación +:

```
hola = 'Hola'
python = 'Python'
hola_python = hola + ' ' + python  # concatenamos 3 strings
print(hola_python)
```

```
Hola Python
```

---

salida

Operadores lógicos o booleanos

A la hora de operar con valores booleanos podemos utilizar and, or y not.

Operador	Descripción	Ejemplos	Resultado
And	Si y sólo si todos los elementos son **True** dará por resultado True. Sino False	True and True False and False True and False False and True	True True False False
Or	Si algún elemento es **True** dará por resultado True. Sino False	True and True False and False True and False False and True	True False True True
Not	El operador not es unario, de negación, solo dará **True** si su elemento es **False** y viceversa.	not True not False	False True

Prioridad de los operadores en Python

Al igual que ocurre en las matemáticas, los operadores en Python tienen un orden de prioridad. De mayor a menor prioridad el orden es el siguiente, :

1. aritméticos (con el mismo orden de prioridad que en las matemáticas).
2. operadores de comparación y pertenencia
3. operadores lógicos
4. asignación

Este orden se puede alterar con el uso de los paréntesis ().

BIBLIOGRAFÍA

[1] Python para todos. R. Gonzalez Duque. Licencia Creative Commons Reconocimiento 2.5 España.

Descarga gratuita de versión más reciente en http://mundogeek.net/tutorial-python/

[2] Python 3 al descubierto 2da Edición. A. Fernández Montoro. Alfaomega. Año 2013