SEAS
CAMPUS SEAS

2

UNIDAD DIDACTICA 2

Python

2. Tipos de datos

ÍNDICE

OBJETIVOS

- Conocer los tipos de datos existentes en Python.
- Comprender la forma de utilizar distintos tipos de datos.
- Conocer su representación interna y externa.
- Conocer el rango de valores que pueden tomar, y las operaciones que se pueden realizar entre variables de los distintos tipos.
- Aprender a aplicar los tipos de datos en función de la información con la que trabaja el programa.

INTRODUCCIÓN

En general, una variable solo puede tomar valores dentro de una “clase de valores”. En programación, a las distintas clases de valores que se pueden utilizar se les denomina “tipos”.

El concepto de tipo de datos es algo abstracto e independiente de los símbolos concretos que se empleen para representar los valores. Por ejemplo, los meses del año se pueden representar mediante enteros del 1 al 12. ¡Pero los meses del año no son enteros!. No tiene sentido sumar meses.

En general, se habla de “Tipos abstractos de datos”, que identifican tanto el conjunto de valores que pueden tomar los datos de cierto tipo, como las operaciones que pueden ejecutarse sobre dichos valores. Como características de los tipos de datos, se pueden citar:

- Cada tipo de datos tiene una *representación interna* en el ordenador que corresponde a la forma en que cada dato se almacena en memoria.
- Cada tipo de datos tiene un *dominio* de posibles valores que los datos pueden tomar.
- Cada tipo tiene una *representación externa* que corresponde a la forma que tienen los datos al introducirlos por el teclado o de presentarlos en la pantalla.
- Cada tipo de datos tiene unas operaciones que se pueden realizar con los datos de ese tipo.

En esta unidad se van a estudiar las características de los tipos de datos de que dispone Python y se practicará con ellos al objeto de dominar su representación, las operaciones que se pueden realizar con ellos y su aplicación en función del formato de la información con la que tratan.

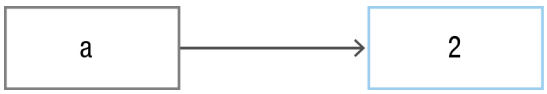
2.1. Clasificación de los tipos de datos

Hablando en términos generales, en Python los tipos de datos pertenecen a una de las dos clases siguientes:

- Inmutables. Son aquellos cuyo valor no puede variar a lo largo de la ejecución de un programa.
- Mutables. Son aquellos cuyo valor se puede cambiar en tiempo de ejecución.

La mutabilidad y la inmutabilidad tiene que ver con la forma en que se representan en Python los datos en memoria. Conviene aclarar este concepto que no se da en otros lenguajes de programación tal y como se hace en Python. En Python todo son objetos y, por ejemplo, bajo esa premisa el número 2 es un objeto. Obviamente el número 2 es siempre el número 2 y no puede cambiar, por lo que es un objeto inmutable. Python está diseñado para consumir la menor cantidad de memoria posible, por lo que únicamente almacena una vez en memoria cada objeto inmutable. Si dos o más variables contienen ese valor, aparece una sola vez en memoria y sus referencias apuntan a esa zona de memoria tal como se muestra en la figura 2.1. Por ejemplo:

```
>>> a= 2
```




```
>>> b= 1+1
```

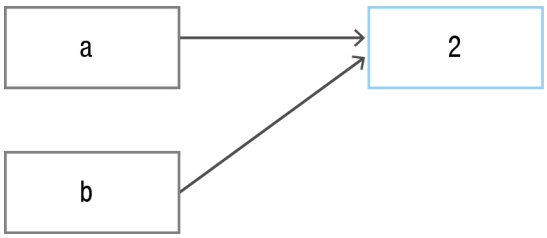
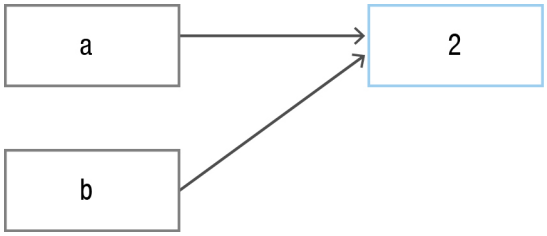


Figura 2.5. Superior, una variable (referencia) apunta al objeto que contiene el número 2. Inferior, dos variables apuntan al mismo objeto,.

a y b son variables enteras pero, en realidad son referencias a la zona de memoria en la que se encuentra el número 2. Si tras asignar 2 a la variable a, se asigna 2 a la variable b, se crea una nueva referencia a la zona de memoria en la que se encontraba el 2 anterior.

Cuando se modifica una variable inmutable lo que sucede es que la correspondiente variable pasa a apuntar a otra zona de memoria que contiene el nuevo valor. En la figura 2.2. se muestra cómo se crea una referencia nueva al asignar el valor 3 a la variable b.

```
>>> a= 2
>>> b= 1+1
```




```
# cambiamos el valor de b
>>> b= 3
```

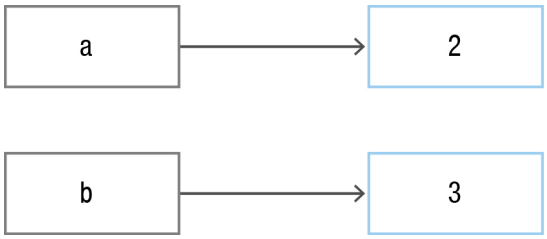


Figura 2.6. Inferior: Tras asignar a "b" el valor 3 se crea una nueva referencia.

¿Qué pasaría ahora si se cambiase el valor de a? Evidentemente ya no habría ninguna referencia al objeto que contiene el número 2, por lo que el sistema lo suprimiría automáticamente sin necesidad de hacerlo el programador.

Dentro de los inmutables cabe hablar de dos clases de tipos:

- **Simples.** Toman un valor único. Son *int, float, complex, bool* y *char*.
- **Secuencias inmutables.** Son agrupaciones de elementos de datos simples. Son *str, unicode, tupla, xrange* y *frozenset*.

A su vez, dentro de los tipos mutables hay tres clases: secuencias mutables, conjuntos mutables y mapeos:

- **Secuencias mutables:** *list* y *range*.
- **Conjuntos mutables**: *set*.
- **Mapeos:** *dict*.

A continuación se representa un cuadro en el que se muestra la clasificación anteriormente citada de los tipos de datos de Python.

Categoría I	Categoría II	Nombre de los tipos
Inmutables	Simples	*int, lfloat, complex, bool* y *char.*
	Secuencias inmutables	*str, unicode, tupla, xrange, frozenset*
Mutables	Secuencias mutables	*list, range*
	Conjuntos mutables	*set*
	Mapeos	*dict*

Figura 2.7. Clasificación general de los tipos de datos.

La función **type**(x) devuelve el tipo correspondiente a la variable x.

EJEMPLO

Por ejemplo, si x fuese entero, se tendría:

```
>>> x = 20
>>> print(type(x))
'int'
```

2.2. Tipos inmutables

En este apartado se van a ver los tipos inmutables simples y las secuencias inmutables.

2.2.1. Tipos inmutables simples

Vamos a comenzar por estudiar los tipos inmutables simples existentes en Python. Se caracterizan por la existencia de una relación de orden total entre sus componentes. Esta relación de orden total, permite la comparación entre los valores. Para ello se usan los operadores de relación, que son:

Operadores de relación en Python

==	Igual que
!=	Distinto de
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que

El resultado de estas operaciones es: verdad (*True*) o falso (*False*).

EJEMPLO

Por ejemplo:

```
>>> a, b, c = 3, 3, 5
>>> a==b  # el resultado es True (verdad) ya que a es igual que b
True
>>> a>c   # el resultado es False (falso) ya que a no es mayor que c
False
>>> a!=c  # el resultado es True (verdad) ya que son distintos.
True
```

Los tipos entero, booleano y carácter son enumerables. Por el contrario, el tipo real es matemáticamente no enumerable, ya que entre dos reales cualesquiera hay infinitos reales. Evidentemente esto hay que matizarlo en informática, ya que un ordenador no puede representar infinitos números. Hay que cuidar sobre todo con los operadores == y !=, ya que dos números muy próximos pero distintos pueden tener la misma representación interna.

En el siguiente ejemplo, a es distinto de b. Sin embargo, como son números tan próximos el ordenador los toma como si fuesen iguales.

EJEMPLO

```
>>> a = 2.0/3.0
>>> b = 2.00000000000000000001/3.0
>>> print(a)
0.6666666666666666
>>> print(b)
0.6666666666666666
>>> a == b
True
```

2.2.1.1. El tipo booleano

Una variable de tipo booleano únicamente puede tomar los valores lógicos *True* (verdad) o *False* (falso).

EJEMPLO

Por ejemplo:

```
>>> coche=True
>>> casa=False
```

asignan a la variable coche el valor lógico *True* y a la variable casa el valor *False*.

Hay otros valores literales que se pueden usar como valores booleanos, como por ejemplo un entero, un carácter, una cadena, etc. Para ello se usa el operador *bool*(), que convierte su argumento en un valor booleano.

EJEMPLO

Por ejemplo:

```
>>> bool(1)=True
>>> bool()=False
>>> bool(-1)=True
>>> bool('ensayo')=True
>>> bool(15)=True
```

Sobre datos de tipo booleano se pueden aplicar muchos de los operadores que se estudian en el álgebra de Boole.

Para representar el resultado de aplicar un operador se pueden usar las Tablas de Verdad. Una tabla de verdad es una representación del resultado de un operador lógico (o de una función lógica, llamada predicado), para todas las posibles combinaciones de los operandos. Es una forma de describir los operadores booleanos y las expresiones lógicas.

Los operadores booleanos de que dispone Python son: *not* (Negación), *and* (producto lógico) y *or* (suma lógica), y responden a las siguientes tablas de verdad:

not	Entrada	Resultado
	falso	verdad
	verdad	falso

Como se ve, la salida es el contrario de la entrada. Se dice que la salida es la entrada negada. El operador *or*, también llamado "suma lógica", responde a la siguiente table de verdad:

or	Operando 1	Operando 2	Resultado
	falso	falso	falso
	falso	verdad	verdad
	verdad	falso	verdad
	verdad	verdad	verdad

Finalmente, el operador and, también llamado "producto lógico", responde a la siguiente tabla de verdad:

and	Operando 1	Operando 2	resultado
	falso	falso	falso
	falso	verdad	falso
	Verdad	falso	falso
	Verdad	verdad	verdad

EJEMPLO

Por ejemplo:

```
>>> not True
False
>>> True and True
True
>>> True and False
False
```

Vamos a recordar las leyes de Morgan que suelen estudiar en los Sistemas Lógicos.

```
not(a and b) = (not a)  or (not b)

not(a or b) = (not a) and (not b)
```

Se puede enunciar diciendo que e negado del producto es la suma de negados, y que el negado de la suma es el producto de los negados.

Los siguientes valores se interpretan como:

- falso: *False*, 0, *None*, cadenas de caracteres vacías, contenedores incluyendo cadenas de caracteres, tuplas, listas, diccionarios y conjuntos.
- verdad: el resto de valores se interpretan como verdad.

Una función interesante es la or-exclusiva, no implementada en Python, también designada como *xor*. Su tabla de verdad es:

xor	Operando 1	Operando 2	Resultado
	falso	falso	falso
	falso	verdad	verdad
	verdad	falso	verdad
	verdad	verdad	falso

¿Cómo se obtiene una expresión lógica para expresarla en términos de las funciones *and* y or?. Veamos para la función *xor*. Partiremos de la tabla de verdad anterior, y nos fijamos en aquellas combinaciones que tienen resultado verdad. En la tabla anterior son dos por lo que darán lugar a una expresión de una suma lógica (*or*). Si llamamos a los operandos x e y se puede escribir, sustituyendo verdad por la variable sin negar y falso por la variable negada:

```
>>> xor = (not(x) and y) or (x and not(y))
```

2.2.1.2. Tipo entero

- Representación externa. La representación externa de los datos de tipo entero es: pueden comenzar con el signo (si es positivo no es necesario) seguido de una secuencia de dígitos numéricos. Cuando se escribe un número en sistema decimal cada dígito numérico es un número del 0 al 9.
- Representación interna. La representación interna de los números enteros se realiza en binario en complemento a dos.

Hay un tipo entero en Python, que es *int*.

EJEMPLO

Como ejemplo, se puede escribir:

```
>>> e=8
>>> print (e), type (e)
8
<type  'int'>
```

El literal *int* se pueden codificar en binario, decimal, hexadecimal u octal. Para escribir un número en binario basta con que su valor comience por 0b. En hexadecimal basta con que su valor comience por 0x; para escribirlo en octal, debe comenzar por 0o.

Con respecto a los operadores aritméticos sobre números enteros en Python, son:

Operador	Operación	Ejemplo
+	Suma	5+25
-	Resta	23-5
*	Multiplicación	63*2
//	División entera	32/3 # el resultado sería 10
%	Módulo (resto)	32%3 # el resultado sería 2
**	Potenciación	3**2 # el resultado sería 9

Como veremos más adelante, estos mismos operadores se pueden usar con números reales, salvo el operador de división entera y el operador módulo. Al utilizar los operadores aritméticos, Python devuelve un resultado del mismo tipo que el operando de mayor precisión. Por ejemplo, al multiplicar un entero por un real, el resultado es de tipo real.

- Como ya hemos visto, y repetimos aquí, los operadores de relación son:

Operador en Python	Ejemplo	Significado
==	x==y	x es igual que y
!=	x != y	x distinto de y
>	x>y	x mayor que y
<	x<y	x menor que y
>=	x>=y	x mayor igual que y
<=	x<=y	x menor o igual que y

Dado que en los enteros existe una relación de orden se les pueden aplicar los operadores relacionales, dando un resultado booleano.

EJEMPLO

Por ejemplo:

```
>>> x, y = 4, 8
>>> x == y
False
>>> x < y
True
>>> x != y
True
```

- Los enteros se representan en memoria (como toda información en el ordenador) mediante un conjunto de unos y ceros. Podemos pensar en la existencia de operadores binarios, que operan sobre enteros en Python. Son aquellos que realizan operaciones booleanas bit a bit sobre números enteros. Si suponemos que se declaran:

```
>>> a = 6    # binario  00000000 00000000 00000000 00000110
>>> b = 4    # binario  00000000 00000000 00000000 00000100
```

Operador	Operación	Ejemplo	Resultado
|	OR bit a bit	c = a|b	00000000 00000000 00000000 00000110
&	AND bit a bit	c = a & b	00000000 00000000 00000000 00000100
^	OR-EXCLUSIVA bit a bit	c = a ^ b	00000000 00000000 00000000 00000010
~	COMPLEMENT bit a bit	c = ~ a	11111111 11111111 11111111 11111001
>>	Desplazamiento a la derecha	c = a >> 2	00000000 00000000 00000000 00000001
<<	Desplazamiento a la izquierda	c = a << 2	00000000 00000000 00000000 00011000

En el desplazamiento a la derecha, entra un uno o un cero por el bit más significativo según sea el bit de signo. En el desplazamiento a la izquierda, entra un cero en el bit menos significativo.

2.2.1.3. El tipo carácter

Incluye como valores todos los caracteres disponibles en un computador.

- Representación externa: como ya se ha visto, los caracteres se escriben entre comillas simples o dobles dentro del programa.
- Representación interna: internamente se asocia un código de unos y ceros para codificar el conjunto de caracteres. Habitualmente se usaba el código ASCII (American Standard Code for Information Interchange).

Inicialmente el código ASCII codificaba en 7 bits, de 0 a 127, los caracteres anglosajones. El 65 representa la "A" mayúscula y el 97 la "a" minúscula. El resto de los caracteres van ordenados según el abecedario. Los idiomas del oeste de Europa como el español, el francés y el alemán utilizan un sistema de codificación llamado ISO-8859-1 (también llamado "latin-1") que extiende el espacio de códigos anterior de 128 a 255 utilizando 8 bits. Finalmente Unidode extiende los códigos para realizar una representación con 16 bits de forma que se puedan codificar caracteres de otros idiomas. A continuación se da una tabla de los códigos correspondientes a los caracteres en ASCII:

Decimal		Decimal		Decimal		Decimal		Decimal		Decimal	
0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ
6	ACK	7	BEL	8	BS	9	HT	10	LF	11	VT
12	FF	13	CR	14	SO	15	SI	16	DLE	17	DC1
18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS
30	RS	31	US	32	SPC	33	!	34	"	35	#
36	$	37	%	38	&	39	`	40	(	41	)
42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5
54	6	55	7	56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?	64	@	65	A
66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M
78	N	79	O	80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W	88	X	89	Y
90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e

Decimal		Decimal		Decimal		Decimal		Decimal		Decimal	
102	f	103	g	104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o	112	p	113	q
114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}
126	~	127		160		161	¡	162	¢	163	£
164	¤	165	¥	166	¦	167	§	168	¨	169	©
170	ª	171	«	172	¬	173		174	®	175	
176	°	177	±	178	²	179	³	180	´	181	µ
182	¶	183	·	184	¸	185	¹	186	º	187	»
188	¼	189	½	190	¾	191	¿	192	À	193	Á
194	Â	195	Ã	196	Ä	197	Å	198	Æ	199	Ç

Decimal		Decimal		Decimal		Decimal		Decimal		Decimal	
200	È	201	É	202	Ê	203	Ë	204	Ì	205	Í
206	Î	207	Ï	208	Ð	209	Ñ	210	Ò	211	Ó
212	Ô	213	Õ	214	Ö	215	×	216	Ø	217	Ù
218	Ú	219	Û	220	Ü	221	Ý	222	Þ	223	ß
224	à	225	á	226	â	227	ã	228	ä	229	å
230	æ	231	ç	232	è	233	é	234	ê	235	ë
236	ì	237	í	238	î	239	ï	240	ð	241	ñ
242	ò	243	ó	244	ô	245	õ	246	ö	247	÷
248	ø	249	ù	250	ú	251	û	252	ü	253	ý
254	þ	255	ÿ								

Figura 2.8. Correspondencia entre los códigos ASCII y los enteros.

- El código Unicode. Cada día es más importante que los desarrolladores de *software* abarquen a los distintos mercados internacionales, y que se intente reducir el desfase de tiempo entre la distribución de las versiones en un determinado país y las internacionales. Para ello, una aplicación puede obtener información específica del país (del sistema operativo, así en Windows se pueden examinar los valores del Panel de Control para determinar las preferencias del usuario).

  El problema surge del hecho de que algunos lenguajes tienen tantos símbolos distintos que no se pueden codificar con un *byte* (8 *bits*). Para ello, se han creado los conjuntos de caracteres de dos *bytes* (16 *bits*). Son complicados de programar porque algunos caracteres son de un *byte* y otros de dos bytes.

Unicode es un estándar fundado por Apple y Xerox en 1988. En 1991, se creó un consorcio para desarrollar y promocionar Unicode. Estaba formado por Adobe, Aldus, Apple, Borland, Digital, Go, IBM, Lotus, Metaphor, Microsoft, NeXT, Novell, el Research Libraries Group, Sun, Taligent, Unisys, WordPerfect y Xerox.

Unicode ofrece una manera simple y consistente de representar caracteres. Todos los caracteres Unicode son de 16 bits (2 *bytes*). Por esta razón, se dispone de más de 65 000 caracteres para codificar todos los caracteres de todos los lenguajes del mundo.

Actualmente están definidos códigos para los alfabetos árabe, chino, bopomofo, cirílico (ruso), griego, hebreo, japonés kana, etc. También se incluyen una gran cantidad de símbolos de puntuación, símbolos matemáticos, flechas, etc. Juntando todo esto, suman unos 34 000 caracteres, lo que deja libres casi la mitad para ampliaciones futuras.

Los códigos se dividen por regiones, algunas de las cuales se muestran en la tabla:

Código de 16 bits	Caracteres
0000-007F	ASCII
0080-00FF	Caracteres latín 1
0100-017F	Latín europeo
0180-01FF	Latín ampliado
0370-03FF	Griego
0600-06FF	Árabe

Las ventajas de utilizar Unicode son:

- Habilita un intercambio de datos sencillo entre idiomas.
- Permite la distribución de un solo archivo EXE o DLL binario que admite todos los lenguajes.
- Mejora la eficiencia de la aplicación.

Python puede utilizar el formato Unicode para la representación de los caracteres, por lo que los codifica con 2 *bytes*. Para ello, se antepone una u.

EJEMPLO

Por ejemplo:

```
>>> car = u'A'
>>> car
'A'
print(car)
A
```

Las operaciones disponibles sobre los datos de tipo caracter son:

- Operadores relacionales: == , !=, <, <=, >, >=
- Las funciones *chr* y *ord*, que convierten respectivamente un entero en el carácter correspondiente al código ASCII codificado mediante ese entero y viceversa para la función *ord*.

EJEMPLO

Por ejemplo:

```
>>> chr(65)
'A'
>>> ord(u'A')
65
```

El carácter Unicode para la "ñ! Es U+0xf1 ex hexadecimal, es decir 241 en decimal, que se puede escribir en la forma: \xf1.

```
>>> ord(u'ñ')
241
```

Los operadores relacionales se aplican sobre la relación de orden de la codificación ASCII.

EJEMPLO

Por ejemplo:

```
>>> a<b
True
```

Ya que la letra "a" se codifica con el entero 97 y la "b" con el entero 98. Hay que tener cuidado ya que la "B" tiene código 66, por lo que:

```
>>> 'B'<'b'

True
```

Con las letras acentuadas también hay que tener cuidado ya que tienen códigos mayores que las letras sin acentuar. Lo mismo se puede decir de caracteres como la "ñ", "ä", "ë", etc.

2.2.1.4. El tipo real

Los tipos reales aproximan la representación de los números reales y corresponden a los números que tienen punto decimal. En virtud de la *Potencia del continuo*, entre cada dos reales existen infinitos números reales, razón por la cuál se necesitarían infinitos bits para poder representar los números reales. Por esto, se hace una representación aproximada (la más próxima con el número de bits y el formato empleado). Para la representación de los reales se usa la notación en coma flotante (*floating point*, en la literatura anglosajona), de forma que las variables de tipo real se representan mediante dos enteros: mantisa y exponente. La resolución dependerá de la mantisa y el alcance (valores mayor y mas próximo a cero) del exponente. La representación externa de los datos de tipo real se realiza escribiendo primero la parte entera, seguida del punto y a continuación la parte decimal.

EJEMPLO

Por ejemplo:
pi=3.1416

Las operaciones aritméticas predefinidas con los reales son: suma (+), resta (-), producto (*), cociente real (/) y cociente entero (//). Debemos aclarar que basta con que un operando sea real para que el operador // opere sobre reales dando como resultado un número real, cuyo valor coincidirá con el del cociente entero. Por ejemplo, vamos a considerar:

```
x = 7.2
y = 3.4
print(x//y)
x = 7
y = 3
print(x//y)
```

En el primer caso el resultado es 2.0 de tipo real. En el segundo es 2 de tipo entero.

Además de los operadores aritméticos también se les puede aplicar los operadores relacionales, cuidando con el operador ==. Veamos el siguiente ejemplo:

```
print(1/3)
print(1/3 == 0.33333333)
```

El resultado es:

```
0.3333333333333333
False
```

En Python hay un único tipo real que es el *float*. En otros lenguajes de programación también existe el tipo *double*, no en Python. Python implementa el tipo *float* a bajo nivel mediante una variable de tipo *double* de C, por lo que utiliza 64 *bits* para la representación. Utiliza un bit para el signo, 11 para el exponente y 52 para la mantisa.

Tipo	Bytes	Rango
float	8	$\pm 1{,}7976931348623157 \times 10^{308}$ $\pm 2{,}2250738585072020 \times 10^{-308}$

También se puede usar notación científica empleando el carácter "e" como exponente (diez elevado a).

EJEMPLO

Por ejemplo:

```
>>> r1=2.3123
>>> r2=0.3e-3
```

Tras realizar una operación con tipos diferentes, el intérprete determina el tipo del resultado tomando el tipo más preciso de los operandos. Por ejemplo, una multiplicación de una variable de tipo *float* con una variable de tipo *int* da un resultado de tipo *float*. Tras una operación entre un operando entero y un flotante, el resultado es del tipo del operando flotante.

La función *round* devuelve un redondeo del número real con un determinado número de decimales que se pasa como argumento.

EJEMPLO

Por ejemplo, si se quiere redondear el valor de r1 con dos decimales, se escribe:

```
>>> r = round(r1, 2)
>>> print(r)
2.31
```

2.2.1.5. El tipo complejo

Como ya sabemos, un número complejo es aquel que tiene parte real y parte imaginaria. El tipo complejo, denominado *complex* en Python se utiliza mucho en ingeniería.

A diferencia de otros lenguajes de programación, en Python un número complejo se escribe:

```
<real>+<imag>j
```

Donde <real> e <imag> designan respectivamente la parte real e imaginaria del complejo, que son números reales (*float*). No se pone ningún símbolo entre la "j" y la parte imaginaria. Para acceder a la parte real e imaginaria de un complejo se usa:

```
complejo.real

complejo.imag
```

EJEMPLO

```
>>> c1 = 5.3+2.6j
>>> print (c1, c1.imag, c1.real, type(c1))      #   es
obligatorio el paréntesis.
(5.3+2.6j) 2.6 5.3 <class 'complex'>
```

2.2.1.6. Conversión de tipos

Es el proceso de convertir valores de un tipo de datos en otro tipo. En Python hay dos tipos de conversión: implícita y explícita.

En la implícita un dato de cierto tipo se convierte automáticamente en otro dato de un tipo distinto.

EJEMPLO

Por ejemplo:

```
>>> num_int = 250
>>> num_flot = 2.50
>>> num = num_int + num_flot
>>> print("El tipo de num_int es: ", type(num_int))
El tipo de num_int es:  <class 'int'>
>>> print("El tipo de num_flot es: ", type(num_flot))
El tipo de num_float es:  <class 'float'>
>>> print("Valor de num= ", num)
El valor de num es:  252.5
>>> print("El tipo de num es: ", type(num))
El tipo de num es:  <class 'float'>
```

En este ejemplo se suma una variable entera y una variable real. La variable num tiene que ser real para que no haya pérdidas de la parte decimal.

EJEMPLO

Vamos a sumar un número entero y una cadena de caracteres numéricos.

```
>>> num_int = 250
>>> num_str = ="789"
>>> print(num_int+num_str)
```

Evidentemente nos va a dar error ya que tal suma no se puede realizar.

Para realizar una conversión explícita, existen funciones predefinidas tales como: *int*(), *float*(), *complex*(), *str*() que realizan la conversión a *int, float, complex* y *str* respectivamente.

EJEMPLO

```
>>> ent1 = 250
>>> str2  =”789”
>>> ent2 = int(str2)    # convierte la cadena en un
número entero
>>> ent=ent1+ent2 # ahora ya se pueden sumar
>>> print(ent1+int(str2))
1039
```

Las funciones de conversión de tipos en Python son las siguientes:

Funciones de conversión	
float()	Convierte un parámetro entero en real.
int()	Convierte un parámetro real en entero.
str()	Convierte un parámetro numérico en una cadena de caracteres.
bin()	Convierte un parámetro entero en una cadena correspondiente al número en binario.
oct()	Convierte un parámetro entero en una cadena correspondiente al número en octal.
hex()	Convierte un parámetro entero en una cadena correspondiente al número en hexadecimal.
round()	Redondea al entero mas próximo. Si se usa con dos parámetros, el segundo indica el número de dcimales que se van a preservar.
complex()	Transforma un número real a complejo con parte imaginaria 0.

2.2.2. Secuencias inmutables

En este apartado se van a ver las secuencias inmutables, que pueden ser cadenas de caracteres, tuplas, rangos inmutables y conjuntos inmutables.

2.2.2.1. El tipo cadenas de caracteres

Una cadena de caracteres **str** (de *string*, cadena) es una sucesión de caracteres. Son secuencias inmutables, por lo que no pueden cambiar en tempo de ejecución. Su representación interna representa los caracteres en código binario en el mismo orden en que aparecen. Los caracteres pueden estar codificados en ASCII o Unicode. Su representación externa representa la cadena entre comillas simples o dobles.

Para cadenas largas que pueden ocupar más de una línea se pueden usar tres comilas dobles al principio y al final.

```
>>> cadLarga = """Esta es una cadena larga que ocupa tres
líneas.

La primera ya se ha escrito, la segunda es esta.

Termino con la tercera línea"""
```

Cuando la variable cadLarga se usa como argumento de la función **print**, se escriben en pantalla las mismas tres líneas tal como se han escrito anteriormente. También se puede hacer con una contra-barra al final de las dos primeras líneas.

EJEMPLO

Por ejemplo:

```
>>> cadLarga = 'Esta es una cadena larga que ocupa
tres líneas.\
La primera ya se ha escrito, la segunda es esta.\
Termino con la tercera línea'
```

Una cadena con comillas simples al principio debe terminar con comillas simples. Una con comillas dobles debe terminar en comillas dobles. Una cadena entre comillas simples puede contener comillas dobles y una entre comillas dobles puede contener comillas simples. Si se quieren contener comillas simples dentro de una cadena entre comillas simples hay que usar la contrabarra en la forma \'. Lo mismo con dobles comillas \".

```
>>> print ("Cadena con 'palabra' entre comillas simples")
```

Cadena con 'palabra' entre comillas simples.

```
>>> print ('Cadena con "palabra" entre comillas dobles')
```

Cadena con "palabra" entre comillas dobles.

```
>>> print ('Cadena con \'palabra\' entre comillas simples')
```

Cadena con 'palabra' entre comillas simples.

```
>>> print ("Cadena con \"palabra\" entre comillas dobles")
```

Cadena con "palabra" entre comillas dobles.

Como la contra-barra sirve como símbolo de escape, para embeber en una cadena un literal contra-barra se deben usar dos contra-barras.

```
>>> fich='C:\\Users\\Antonio'

>>> print(fich)

C:\Users\Antonio
```

Para las cadenas largas se usa la representación entre tres comillas simples o dobles.

- Prefijo de cadenas de caracteres. Una cadena de caracteres puede ir precedida por un carácter:
  - r/R, que indica que se trata de una cadena *raw* (cruda). Se distinguen de las normales en que los caracteres de escape mediante la barra invertida (\) no se sustituyen por sus contrapartidas.
  - u/U, que indica que se trata de una codificada en Unicode.

Cuando se escribe una cadena de caracteres Python intenta convertirla en código ASCII. Si la cadena no tiene caracteres que no sean ASCII, dado que la cadena Unicode está constituída por caracteres que son igualmente caracteres ASCII, su escritura produce el mismo efecto y no hay diferencia.

EJEMPLO

Veamos un ejemplo en que una cadena tiene caracteres que no son ASCII, por ejemplo, una “ñ”.

```
>>> s = u'La Pe\xf1a'
>>> print (s)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in
range(128)
>>> print s.encode('latin-1')
La Peña
```

EJEMPLO

Ejemplo:

```
>>> print 'Esta cadena contiene una contra-barra: \\'
Esta cadena contiene una contra-barra: \
>>> print r'Esta cadena contiene una contra-barra: \'
Esta cadena contiene una contra-barra: \
```

EJEMPLO

Ejemplo:

- En código ASCII:

```
>>> 'Hola Mundo'
'Hola Mundo'
```

- En Unicode:

```
>>> u'Hola Mundo'
'Hola Mundo'
```

A. Operaciones sobre cadenas de caracteres

El hecho de que las cadenas de caracteres sean inmutables condiciona las operaciones que se puedan efectuar con ellas. Por ejemplo, una asignación a un componente de la cadena (es decir, a un carácter previamente existente) de la cadena está prohibida, y si se intenta, se genera un error de tipo. Para leer un determinado carácter de una cadena se pasa su índices entre corchetes separados por dos puntos, teniendo en cuenta que el primer carácter de la cadena tiene índice cero.

EJEMPLO

Ejemplo:

```
>>> a = 'Hola Mundo'
>>> a[0]  # lee la primera componente de la cadena
>>> 'H'
>>> a[0] = 'h'      # operación prohibida
Traceback (most recent call last):
    File "<input>", line 1, in <module>
TypeError:  'str'  object  does  not  support  ítem
assignement
>>> a[-1]
o
```

Vemos que cuando el índice es negativo se comienza por el final de la cadena.

Se pueden obtener los caracteres comprendidos entre dos índices escribiendo los índices entre corchetes. Por ejemplo a[i:j] obtiene los caracteres comprendidos entre los índices i y j-1. Para el caso en que se desee una subcadena que comience por el primer carácter, no es necesario poner el 0. Cuando se desee una que llegue hasta el final desde una cierta componente, no es necesario poner la última.

EJEMPLO

Por ejemplo:

```
>>> a1 = a[:3]
Hol
>>> a2 = a[5:]
Mundo
```

Hay tres operaciones fundamentales sobre cadenas de caracteres:

- Concatenación. Se usa el operador "+". Da lugar a una cadena compuesta por las dos cadenas que se concatenan en el mismo orden en que van escritas.

```
>>> b = 'Saludo: '

>>> b+a

Saludo: Saludo: Hola Mundo
```

- Multiplicación. Se usa el operador "*". Se repite la primera cadena (izquierda del operador *) tantas veces como indica el valor numérico entero que figura a la derecha del operador *.

```
>>> a*2

'Hola MundoHola Mundo'
```

- Interpolación. Se usa el operador %. Se utiliza para formatear. Va seguido por el tipo que necesita ser convertido. Hay que determinar el tipo del objeto:

%c	str	Convierte a carácter
%s	str	A cadena de caracteres
%d	int	Enteros con signo
%f	float	Reales
%o		Octal
%x		Hexadecimal

```
>>> print( "Mi nombre es %s y mi peso es %d Kg" % ('Antonio', 43))
```

Mi nombre es Antonio y mi peso es 43 Kg.

Vemos que se sustituye %s por la cadena de caracteres y %d por el número entero.

Se pueden aplicar los operadores relacionales a las cadenas de caracteres. Se comenzará comparando el primer carácter, hasta llegar a uno que sea mayor en una cadena que en la otra y que dará lugar a considerar la cadena mayor.

```
>>> 'Banco' > 'bahia'

False
```

Ya que 'B'<'b'.

Existen una serie de funciones que se aplican sobre cadenas de caracteres. Vamos a ver algunas de ellas.

- La función len() devuelve el número de caracteres de una cadena de caracteres.

EJEMPLO

Por ejemplo:

```
>>> len('Hola Mundo')
10
```

- A su vez la función str(arg) convierte el argumento que se le pase en una cadena de caracteres.

EJEMPLO

Por ejemplo:

```
>>> str(5)
'5'
>>> str(2.24)
'2.24'
>>> str(3+8j)
'3+8j'
```

2.2.2.1.1. Códigos de control sobre cadenas de caracteres

Como ya sabemos, los caracteres que aparecen en las cadenas de caracteres con letras, dígitos, símbolos de puntuación y algunos otros símbolos imprimibles. Aparte de estos caracteres pueden contener caracteres especiales conocidos como códigos de control, ya que controlan la forma en la que el texto aparece en la pantalla o en una impresora. El símbolo contra-barra "\" indica que el carácter que le sigue es un carácter de control y se conoce como símbolo de escape. Por ejemplo, \n representa un paso a una nueva línea moviendo el cursor a la línea siguiente. Otros códigos son: \t para separar mediante una tabulación, \f para saltar a otra página en una impresora, \b para un retroceso.

EJEMPLO

Por ejemplo:

```
print('A\nB\nC\nD')
A
B
C
D
print('E\tF\tG\tH')
E   F     G     H
print('WX\bYZ')
WYZ
print('1\a2\a3\a4\a5\a6')
123456
```

También se puede usar la contra-barra como caracteres de una cadena. El siguiente programa usa algunas secuencias de escape comunes.

```
>>> print ("Escritura de apóstrofos: " + "\'A\' " )
```

Escritura entre apóstrofos: 'A'

```
>>> print ("Escritura de comillas: " + "\"cadena\" " )
```

Escritura de comillas: “cadena”

```
>>> print (“Escritura de diagonal invertida: \\” )
```

Escritura de diagonal invertida: \

```
>>> print (“Texto separado\t\tpor dos tabulaciones” )
```

Texto separado por dos tabulaciones.

Se puede usar format para formatear la función print de forma que se sustituyan las componentes de format en la cadena de caracteres que hay en print.

```
>>> cad1 = ‘Tengo’

>>> cad2 = ‘años’

>>> e = 5

>>> print(“Hoy es mi cumpleaños, ya {0} veinte {1} y me han
dado {2} felicitaciones”.format(cad1, cad2, e))
```

Hoy es mi cumpleaños, ya Tengo veinte años y me han dado 5 felicitaciones.

2.2.2.2. Tuplas

Una tupla es una lista indexada de elementos que pueden ser de distintos tipos. En Python el tipo correspondiente se conoce como *tuple*. Es inmutable y por tanto no se puede modificar una vez que se ha creado. Una tupla se crea asignando sus elementos entre paréntesis separados por comas:

```
>>> t = (<elemento_1>, <elemento_2>, …, <elemento_n>)
```

Por ejemplo, la tupla datos se puede crear:

```
>>> datos=(‘Antonio’, 34, ‘Pepe’, 28)
```

Contiene dos cadenas de caracteres y dos enteros. El paréntesis es opcional y bastaría escribir los componentes de la tupla separados por coma.

En el caso de una tupla de un solo elemento, se debe poner la coma a continuación del elemento. Veamos:

```
>>> t = (‘Pepe’)

>>> t

‘Pepe’

>>> t = (‘Pepe’,)

>>> t

>>> (‘Pepe’,)
```

Una tupla vacía se crea con un par de paréntesis o con la función **tuple**():

```
>>> tv = ()

>>> t

()

>>> tuple ()

()
```

Los elementos de una tupla se pueden asignar (desempaquetar) a variables. Por ejemplo, supongamos una tupla t con tres elementos. Se pueden asignar sus elementos a sendas variables x, y, z en la forma:

```
>>> x, y, z = t
```

En donde "x" toma por valor el del primer elemento de la tupla, "y" el segundo y "z" el tercero. Se puede acceder a un lemento de una tupla por medio de un índice escribiéndolo entre corchetes. El índice del primer componente es el cero.

EJEMPLO

Por ejemplo:

```
>>> x=t[0]
```

Se puede acceder a una subtupla usando el operador ":" en la forma:

```
>>> print(t[1:])# escribe desde el segundo elemento
al último.
>>> print(t[:3])# escribe desde el primer elemento al
tercero.
>>> print(t [2:4])    # escribe los elementos de
índice 2 al 4 excluído.
```

Se puede obtener el índice que ocupa un cierto elemento en una tupla t usando la función **index** o el número de veces que aparece un determinado elemento en la tupla mediante **count**.

```
>>> t.index(<elemento_i>)

>>> t.count(<elemento_i>)
```

Las tuplas se usan cuando se quiere usar una secuencia de valores que no se quiere modificar. Por ejemplo, los días de la semana:

```
>>>  semana=('lunes',  'martes',  'miércoles',  'jueves',
'viernes', 'sábado', 'domingo')
```

2.2.2.3. Conjuntos inmutables

En Python existen dos tipos conjunto, uno mutable y otro inmutable. Los conjuntos inmutables se conocen como *frozenset* y los mutables como *set*. Un conjunto es una colección de elementos no repetidos con las operaciones típicas de los conjuntos que se ven en matemáticas: unión, intersección, diferencia, pertenencia, etc. Si se añade algún elemento repetido, las copias se pierden.

Los conjuntos se definen entre llaves con sus elementos separados por comas y no hay una relación de orden definida entre los elementos del conjunto.

EJEMPLO

Por ejemplo, el conjunto muebles se puede definir:

```
>>> muebles = {'silla', 'mesa', 'armario', 'cama', 'trinchante'}
```

El conjunto vacío se define: >>> cv={}

Como se ha comentado, con los conjuntos se pueden realizar las operaciones típicas de unión (|), intersección (&) y diferencia simétrica (^).

Para saber si un determinado elemento pertenece a un conjunto se puede usar el operador **in** (está en, es decir, pertenece) dando resultado booleano. Por ejemplo:

```
>>> s = 'silla' in muebles
True
```

2.2.2.4. Rango mutable

Un rango de longitud n es una sucesión de enteros desde 0 a n-1.

Para generarlos se usa la función range(m,n,p) pasando m, n y p como parámetro. Donde m es el primer valor, n el último y p el paso. Si p no se pone se toma por defecto el valor 1. si no están ni el primero ni el último se supone que parte de 0 y llega a n-1 valor

EJEMPLO

Por ejemplo:

```
>>> range(6) # 0,1,2,3,4,,5
>>> range(3,6) # 3 4 5
    >>> range(3, 6, 2) # 3,5
```

No se imprimen los enteros del rango ni se puede acceder a ellos. Para ver sus valores es necesario convertirlo a una lista:

EJEMPLO

Por ejemplo:

```
>>>x=range(6)
>> print(r)
range(0,6)
>>list(x)
[0,1,2,3,4,5]
```

De amplio uso en las iteraciones (instrucciones que permiten la repetición de un bloque de programación, m-n veces).

```
for i in range (1,10):
  <sentencias>
```

Se puede emplear para crear listas

```
lst = [<expresión> <rango>]
```

EJEMPLO

Por ejemplo:

```
>>> lst1 = [x**2 for x in range(1,6)]
>>> print(lst1)
[1, 4, 9, 16, 25]
Nota: el xrange de Python 2. pasó a ser el range en
Python 3
```

2.3. Tipos de datos mutables

En este apartado vamos a ver los tipos mutables cuyo valor puede cambiar en tiempo de ejecución.

2.3.1. Listas

Muchos lenguajes de alto nivel trabajan con *arrays*, que son la implementación de los vectores en mateméticas. Sin embargo, en Python se trabaja con listas. Los *arrays* son más simples ya que son estructuras de datos de más bajo nivel que no tienen las funcionalidades de las listas.

Una lista es una estructura formada por una secuencia ordenada de elementos. Para asignar una lista a una variable basta con escribir los elementos que forman parte de la lista entre corchetes, separadas pro comas. Las listas son heterogéneas, por lo que los elementos que las componen pueden ser de distinto tipo.

EJEMPLO

Por ejemplo:

```
>>> lst = ['Antonio', 1, 54, 'Fernando', 3, 23]
```

Para acceder a los elementos de una lista se puede usar un índice que se escribe entre corchetes, comenzando por el índice 0 para el primer elemento.

EJEMPLO

Por ejemplo:

```
>>> lst[0]
'Antonio'
>>> lst[2]
54
```

La función len() devuelve el número de elementos de la lista, es decir, su longitud. Se pueden usar índices negativos siendo -1 el índice correspondiente al último elemento. Dado que es mutable, se pueden utilizar los índices para asignar distintos valores a sus componentes.

EJEMPLO

Por ejemplo:

```
>>> lst[0] = 'Federico'
```

Una lista vacía es aquella que no tiene ningún elemento. Se define:

```
>>> lstv=[]
```

De una lista se puede extraer una sublista usando el operador ":" tal como se usa en Matlab. Así, si se escribe:

```
>>> lst = [1, 3, 5, 7, 9, 11, 13, 15]
>>> len(lst)
8
```

Vemos que se crea una lista de 8 componentes, en concreto los ocho primeros números impares. El índice de la lista irá por tanto desde el 0 para el primer componente hasta el 7 para el octavo componente. Para extraer una sublista compuesta por los componentes del tercero (índice 2) hasta el sexto (índice 7) se debe escribir:

```
>>> lst1 = lst[2:8]
```

que asigna a lst1 una lista compuesta por todos los componentes de lst desde el segundo hasta el séptimo. Fijarse que el índice 8 no lo toma pues toma los enteros de 2 hasta 7.

Caso de que no se ponga ningún valor numérico a derecha oizquierda del operador ":" signica que:

- lst(:5) Desde el primero hasta el cuarto componente.
- lst(3:) Desde la componente 3 (la cuarta) hasta la última componente.

Las listas son mutables por lo que se pueden añadir o eliminar componentes. Para añadir un componente al final de la lista se usa el nombre de la lista, un punto y la función **append**:

```
lst.append(componente)
```

Para eliminar un componente por su índice se usa la función **del**:

```
del lst[3]
```

Elimina la cuarta componente de la lista lst.

Existen bastantes funciones que operan sobre listas. A continuación se rsumen algunas de ellas. Otras se verán cuando se studie la programación orientada a objetos.

```
>>> len(lst)  # longitud de la lista
>>> sum(lst)  # suma todos los miembros de la lista
>>> any([1,0,1,0,1])      # hay algún valor "verdad" en la lista
>>> all([1,0,1,0,1])      # son todos los valores "verdad"
```

A su vez, se pueden alicar funciones (mas adelante se llamarán métodos en programación orientada a objetos), como:

```
>>> lst.append(3)    # añade un elemento al final de la lista lst

>>> lst.count(5)     # da el número de veces que un elemento aparece en la lista

>>> lst.extend([2, 4, 6])        # añade varios valores al final de la lista lst

>>> lst.index(5)     # devuelve el índice de un valor

>>> lst.insert(1, 34)      # inserta en la posición 1 el valor 34

>>> x=lst.pop(0).       # elimina un elemento por su índice y lo asigna a la variable x

>>> lst.sort()              # ordena la lista en orden creciente

>>> lst.reverse()    # en orden inverso
```

Se pueden concatenar dos listas usando el operados "+". También se puede concatenar una lista consigo misma un cierto número entero de veces con el operador "*".

2.3.2. Diccionarios

Definen una relación biunívoca entre claves y valores. Es un tipo mutable en el que no existe una relación de orden, por lo que no son aplicables los operadores relacionales. Las claves pueden ser alfabéticas, numéricas o todo tipo que sea similar a una tabla. Basta usar cualquier tipo inmutable incluyendo números y tuplas. Los valores pueden ser numéricos, secuencias, diccionarios, etc.

Para crear un diccionario mediante una asignación basta con crear una lista con pares de valores separados por comas.

EJEMPLO

Por ejemplo:

```
>>> d1 = {"clave1": 135, "clave2": True, "clave3": [1,2,3], "clave4": 340}
>>> print( d1, type(d1))
    {'clave4': 340, 'clave1': 135, 'clave3: [1, 2, 3], 'clave2': True} <type 'dict'>
```

Usando la clave se puede acceder a los elementos de un diccionario así como asignar valores al diccionario:

```
>>> d1['clave1']
```

```
135
>>>  d1['clave2']
True
```

Un diccionario permite eliminar cualquier entrada usando la función **del** en la forma siguiente: **del**(d1[‘clave_2’]).

Se les pueden aplicar algunas funciones. Se comprenderá mejor cuando se estudie la programación orientada a objetos.

EJEMPLO

Por ejemplo:

```
>>> d1.get(clave)           # obtiene un valor a partir de su clave
>>> d1.update(clave: valor) # añade pares (clave,valor) al dicionario
>>> d1.keys()               # muestra todas las claves del diccionario
>>> d1.values()       # muestra todos los valores del diccionario
>>> d1.items()        # muestra todos los pares (clave, valor) del diccionario
```

Por ejemplo:

```
>>> d1.keys()
dict_keys(['clave1', 'clave2', 'clave3', 'clave4'])
>>> d1.values()
dict_values([135, True, [1, 2, 3], 340])
```

Inserción de datos en un diccionario

A partir del índice, basta con asignar el valor correspondiente poniendo el índice entre corchetes. Por ejemplo:

```
dicc = {'a':1, 'b':2, 'd':4, 'e':5}

# insertar un valor a partir de su clave al final del diccionario
dicc['c'] = 3
print(dicc)
```

También se podría crear un diccionario con los pares clave-valor que se desean insertar y utilizar la función update. Por ejemplo:

```
# insertar un par clave-valor usando la función update
d1 = {'f': 6}
dicc.update(d1)
```

```
print(dicc)
```

Ni que decir tiene que se pueden utilizar variables para la clave y el valor. Por ejemplo:

```
clave1 = 'h'
valor1 = 8
dicc[clave1] = valor1
```

2.3.3. Conjuntos mutables

Son colecciones no ordenadas de elementos no repetidos. Se crean mediante la sentencia *set* con un parámetro que es una lista entre corchetes. Así:

```
>>> conj1 = set([4, 5, 7, 11, 8, 9])

>>> print(conj1)

set([4, 5, 7, 11, 8, 9])
```

Operaciones típicas con los conjuntos son la unión (|), intersección (&), suma (+), diferencia (-), diferencia simétrica (^) a las que hay que añadir el incluir un nuevo elemento en un conjunto mutable o el extraerlo.

EJEMPLO

Por ejemplo:

```
>>> conj1 = set([1, 2, 1, 3, 4, 4, 5, 6, 3])  #
1,2,3,4,5,6
>>> conj2 = set([1, 3, 3, 5, 7, 1])           #
1,3,5,7
>>> int = conj1 & conj2            # 1,3,5
>>> uni = conj1 | conj2            # 1,2,3,4,5,6,7
>>> dif = conj1 - conj2            # 2,4,6
>>> dsim = conj1 ^conj2            # 2,4,6,7
```

Existen una serie de funciones que operan con estos tipos de datos pero se verán más adelante.

2.4. Asignación externa

Como se ha comentado, la asignación externa hace uso de periféricos tales como el teclado para asignar valores a las variables y la pantalla del ordenador para mostrar los resultados. A continuación se va a ver la entrada y la salida estándar en Python.

2.4.1. La entrada estandar en python

Cuando se desea introducir datos desde el teclado del ordenador se deben utilizar las funciones integradas en el intérprete de Python. Son:

- eval(input()), que devuelve una lista que se introduce desde el teclado a la variable a la que se le asigna.
- input(), que devuelve una cadena de caracteres que se introduce desde el teclado a la variable a la que se le asigna.

En ambas se pasa como argumento una cadena de caracteres para comunicarse con el usuario acerca de lo que debe introducir. La primera sentencia devuelve como salida una lista y la segunda devuelve una cadena de caracteres.

Ejemplos:

```
>>> nombre=input('Escribe tu nombre: ')

Escribe tu nombre: >? Antonio

>>> nombre

'Antonio


>>> edad=input('Años: ')

Años: >? 23

>>> edad

'23'


>>> ciudades=input("Mayores ciudades de España: ")

Mayores ciudades de España: ["Madrid", "Barcelona",
"Valencia", "Sevilla", "Zaragoza"]

>>> ciudades

"["Madrid", "Barcelona", "Valencia", "Sevilla",
"Zaragoza"]"
```

```
>>> print(ciudades, type(ciudades))

["Madrid", "Barcelona", "Valencia", "Sevilla", "Zaragoza"]
<class 'str'>

>>>

>>> ciudades=eval(input("Mayores ciudades de España: "))

Mayores ciudades de España: ["Madrid", "Barcelona",
"Valencia", "Sevilla", "Zaragoza"]

>>> ciudades

["Madrid", "Barcelona", "Valencia", "Sevilla", "Zaragoza"]

>>> print(ciudades, type(ciudades))

["Madrid", "Barcelona", "Valencia", "Sevilla", "Zaragoza"]
<class 'list'>

>>>

>>> poblacion = input("Población de Zaragoza: ")

Población de Zaragoza: >? 700000

>>> print(población, type(población))

700000 <class 'str'>

>>>

>>> población = int(input("Población de Zaragoza: "))

población de Zaragoza: >? 700000

>>> print(población, type(población))

700000 <class 'int'>

>>>
```

Como vemos, la función **input()** devuelve una cadena de caracteres. Cuando se desea leer un número entero (int) o real (float), hay que convertir la cadena de caracteres que devuelve la función input a entero o real usando funciones de conversión.

EJEMPLO

A continuación se dan algunos ejemplos:

```
>>> x = input('Introduce un entero: ')
Introduce un entero: >? 34
>>> print(x, type(x))
34 <class 'str'>
>>> xe=int(x)
>>> print(xe, type(xe))
34 <class 'int'>
>>> xr=float(x)
>>> print(xr, type(xr))
34.0 <class 'float'>
```

La transformación se puede realizar en una misma línea en la forma:

```
>>> x=float(input('solicitud:')
```

Las listas con cadenas de caracteres numéricas no se pueden transformar en listas numéricas.

EJEMPLO

Por ejemplo:

```
>>> lst=['1', '2', '3']
int(lst)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: int() argument must be a string, a bytes-
like object or a number, not 'list'
```

RESUMEN

- Tipos inmutables son aquellos cuyo valor no varía durante la ejecución del programa. Mutables son los que puede cambiar su valor mientras se ejecuta un programa.
- Tipos simples son los que toman un valor único. Son *int, float, complex* y *bool.* Secuencias son aquel grupo de tipos que toman valores múltiples. Algunas son inmutables y otras mutables.
- Secuencias inmutables son las cadenas de caracteres (*str*), las tuplas (*tuple*), los rangos inmutables (*xrange*) y los conjuntos inmutables (*frozenset*).
- Secuencias mutables. Son las listas (*list*), los rangos (*range*), los diccionarios (*dict*) y los conjuntos mutables (*set*).
- La función type(x) devuelve el tipo correspondiente a la variable x.

Categoría I	Categoría II	Nombre de los tipos
Inmutables	Simples	int, float, complex, bool y char
	Secuencias inmutables	str, unicode, tupla, xrange, frozenset
Mutables	Secuencias mutables	list, range
	Conjuntos mutables	set
	Mapeos	dict

- Operadores relacionales.

Operadores de relación en Python

== Igual que

!= Distinto de

< Menor que

<= Menor o igual que

> Mayor que

>= Mayor o igual que

- Tipo booleano. Una variable de tipo booleano únicamente puede tomar los valores lógicos *True* (verdad) o *False* (falso).
- Los operadores booleanos de que dispone Python son: *not* (Negación), *and* (producto lógico) y *or* (suma lógica).
- Hay dos tipos enteros en Python, que se designan como *int*.

- Operadores con datos de tipo entero:

Operador	Operación	Ejemplo
+	Suma	5+25
-	Resta	23-5
*	Multiplicación	63*2
//	División entera	32/3 # el resultado sería 10
%	Módulo (resto)	32%3 # el resultado sería 2
**	Potenciación	3**2 # el resultado sería 9

- El tipo carácter Incluye como valores todos los caracteres disponibles en un computador. Se escriben entre comillas simples o dobles y se codifican en código ASCII o Unicode en orden alfabético.

- La función *chr* convierte un entero en el carácter cuyo código ASCII es ese entero. La función *ord* hace lo contrario.

- El tipo real es el *float*. Los números reales tienen siempre punto decimal.

- El tipo complejo es *complex*.

- Se puede convertir un tipo en otro mediante las funciones de conversión de tipos. Son:

Funciones de conversión	
float()	Convierte un parámetro entero en real.
int()	Convierte un parámetro real en entero.
str()	Convierte un parámetro numérico en una cadena de caracteres.
bin()	Convierte un parámetro entero en una cadena correspondiente al número en binario.
oct()	Convierte un parámetro entero en una cadena correspondiente al número en octal.
hex()	Convierte un parámetro entero en una cadena correspondiente al número en hexadecimal.
round()	Redondea al entero mas próximo. Si se usa con dos parámetros, el segundo indica el número de dcimales que se van a preservar.
complex()	Transforma un número real a complejo con parte imaginaria 0.

- Una cadena de caracteres **str** (de *string*, cadena) es una sucesión de caracteres. Se escriben entre comillas simples, dobles o triples en cadenas largas.
- La función **len**() devuelve el número de caracteres de una cadena de caracteres.
- Una tupla es una lista indexada de elementos que pueden ser de distintos tipos.
- Una tupla se crea asignando sus elementos entre paréntesis separados por comas.
- Se puede acceder a un elemento de una tupla por medio de un índice escribiéndolo entre corchetes. El índice del primer componente es el cero.
- Los conjuntos inmutables se definen entre llaves con sus elementos separados por comas. No contienen elementos repetidos ni están ordenados.
- Las operaciones típicas son: unión (**|**), intersección (**&**) y diferencia simétrica (**^**).
- Para saber si un determinado elemento pertenece a un conjunto se usa el operador **in**.
- Un rango de longitud n es una sucesión de enteros desde 0 a n-1. Se generan usando la función **range**(m,n), que devuelve una lista con los enteros comprendidos entre m y n excluido.
- Una lista es una estructura heterogénea formada por una secuencia ordenada de elementos.
- Para asignar una lista a una variable basta con escribir los elementos que forman parte de la lista entre corchetes.
- Se puede extraer una sub-lista usando el operador ":" siempre entre corchetes.
- Muchas funciones operan sobre listas: **append, del, len, sum, any** y **all.**
- Un rango mutable es una sucesión aritmética de enteros. La diferencia entre cada dos enteros consecutivos se llama razón o paso. Se pueden crear usando la función **range**() a la que se pasan como parámetros el primer valor, el último-1 y la razón.
- Los diccionarios definen una relación biunívoca entre claves y valores. Las claves pueden ser alfabéticas, numéricas o todo tipo que sea similar a una tabla.
- Los conjuntos mutables se crean mediante la sentencia *set* con un parámetro que es una lista entre corchetes.