DAM/DAW

PROGRAMACIÓN

02 - Python

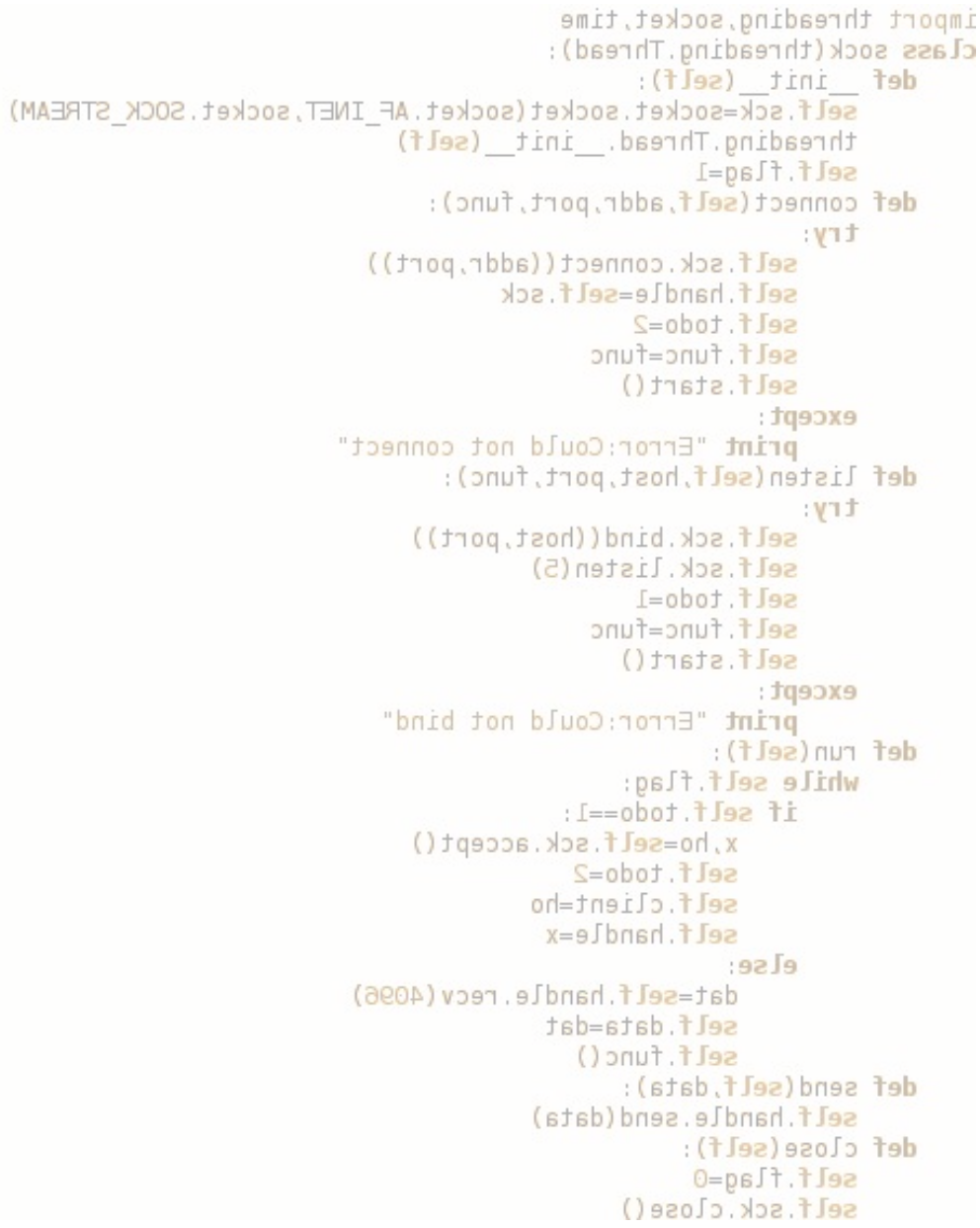
Datos y Operadores

fran@iessanclemente,net

Rev: 3.2

Índice

- Tipos de datos
  - Datos numéricos
  - Valores lógicos
  - Operadores
- Variables
- Cadenas de caracteres
- Funciones predefinidas
- Salida formateada
- Entrada de datos



Tipos de datos

- Con objeto de facilitar el tratamiento de los datos por parte de nuestro programa, Python, al igual que el resto de lenguajes de alto nivel, ofrece una serie de tipos de datos para categorizarlos.
- Estos tipos de datos definen cómo se almacenan internamente dichos datos y qué operaciones podemos realizar con ellos.
- En general, se suele distinguir entre tipos de datos escalares, que se utilizan para tipos de datos atómicos y unidimensionales, y no-escalares, para almacenar datos multidimensionales

Escalares	*No-Escalares*
int, para enteros *float*, para reales (aproximación) *bool*, para valores lógicos (True/False) *NoneType*, para el valor None	*str*, para cadenas de caracteres *tuple*, para tuplas *list*, para listas *dict*, para diccionarios *complex*, para números imaginarios

la función type() permite obtener el tipo de un valor o variable

Datos numéricos (I)

Almacenamiento de números

- En Python se diferencia el almacenamiento de números enteros (int), reales en coma flotante (float) y complejos (*complex*). La librería estándar contiene tipos de datos numéricos adicionales para fracciones (*fraction*) y números decimales de precisión configurable (*decimal*)
- Los números enteros se almacenan con el tipo int y permiten el empleo de valores enteros de cualquier longitud (precisión ilimitada) (Python 3 eliminó la distinción entre los tipos **int** y **long**)
- El tipo *float* utiliza precisión doble (equivalente a ***double*** de C o Java) y dependerá de la arquitectura (usualmente: IEEE-754 *binary64*)
- El tipo *complex* está formado por dos partes, real (*.real*) e imaginaria (*.imag*), cada una de ellas de tipo ***float***

Datos numéricos (II)

- **Ejemplos**

```
>>> 3 + 2
5
>>> 4/2
2.0
>>> type(3.8)
<class 'float'>
>>> 3 + 2 + 4 + 2.0
11.0
>>> type(1 + 3j)
<class 'complex'>
>>> (1 + 3j).imag
3.0
>>> type((1 + 3j).real)
<class 'float'>
```

En Python3, la división siempre genera un *float*

En Python2 no: el tipo del resultado será el de mayor precisión del de los operandos. Si ambos operandos son *int*, el resultado también

- En general, los valores ***int*** suelen ocupar menos espacio de memoria que los de tipo ***float*** y las operaciones son más rápidas

Datos numéricos (III)

- **Aproximaciones decimales**
- Debemos tener presente que, debido al espacio limitado de almacenamiento de los formatos IEEE-754 (por ej., *binary64* → mantisa de 52 bits + 1 implícito), las representaciones de determinados números decimales son aproximaciones. Por ejemplo, los números decimales 0.5 o 0.25, tienen representación exacta en binario (0.1 y 0.01). Sin embargo, para representar el número 0.1_{10} en binario, necesitaríamos infinitos dígitos: 0.000110011001100...
- Esto puede producir algunos resultados inesperados, especialmente al realizar comparaciones:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> (0.1 + 0.2) == 0.3
False
```

Datos numéricos (IV)

- **Operadores aritméticos**

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
División entera	//	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4

- En general, el tipo de dato del resultado será el de mayor precisión de los operandos (en la división siempre es *float*):

```
>>> type(3 + 2)
<class 'int'>
>>> type(3 - 2.0)
<class 'float'>
```

Datos numéricos (V)

- **Precedencia y asociatividad**

  La asociatividad y la precedencia de los operadores establecen la manera en la que se evalúan las expresiones, especialmente cuando intervienen diversos operandos y operadores

```
>>> 3 + 4 - 2
5
```

suma y resta tienen la misma precedencia y se asocian por la izquierda, así que el orden en que se resolvería sería: primero 3+4 y, al resultado, restarle 2

```
>>> 3 + 4 * 2
11
```

el producto tiene mayor precedencia que la suma y se asocia por la izquierda, así que el orden en que se resolvería sería: primero 4*2 y, al resultado, sumarle 3

```
>>> (3 + 4) * 2
14
>>> (3 + 4) * (4 - (2 + 1))
7
```

el uso de paréntesis nos permite modificar el orden en que se evalúa la expresión, desde los paréntesis más internos a los más externos

Datos numéricos (VI)

Representaciones de enteros

- Además de en base 10, podemos representar los literales enteros en bases binaria, octal y hexadecimal. Para indicar una de estas bases, antepondremos al valor numérico el prefijo correspondiente: 0b (para binario), 0o (para octal) y 0x (para hexadecimal)

```
>>> 0b110101
53
>>> 0b1 + 0b0001
2
>>> 0o10 + 0o10
8
>>> 0xcafe
51966
>>> 6 - 0b11*2 + 0xa
10
```

Datos numéricos (y VII)

Operadores a nivel de bit (*Bitwise Operators*)

Operador	Descripción
&	Y (AND)
|	O (OR)
^	O exclusivo (XOR)
~	NOT
>>	Desplazamiento DERECHA
<<	Desplazamiento IZQUIERDA

En decimal		En binario	
Expresión	Resultado	Expresión	Resultado
5 & 12	4	00000101 & 00001100	00000100
5 | 12	13	00000101 | 00001100	00001101
5 ^ 12	9	00000101 ^ 00001100	00001001
5 << 1	10	00000101 << 00000001	00001010
5 << 2	20	00000101 << 00000010	00010100
5 << 3	40	00000101 << 00000011	00101000
5 >> 1	2	00000101 >> 00000001	00000010

Valores lógicos (I)

El tipo *booleano*

- Es habitual en los lenguajes de programación la existencia de un tipo de datos lógico o *booleano*. Python proporciona el tipo *bool* que admite dos posibles valores: True y False. Estos valores se utilizan para representar el resultado de expresiones lógicas (test de verdad)

```
>>> 3 > 2
True
>>> type(7<5 or 6>4)
<class 'bool'>
```

- Hay tres operadores lógicos (de menor a mayor precedencia):
  - or (*"o" lógico*): devuelve True si alguno de sus dos operandos es True
  - and (*"y" lógico*): devuelve True si sus dos operandos son True
  - not (*"no" lógico, negación*): devuelve el valor contrario del operando

Valores lógicos (II)

- **Tablas de verdad**

and		
Operandos		Resultado
True	True	True
True	False	False
False	True	False
False	False	False

or		
Operandos		Resultado
True	True	True
True	False	True
False	True	True
False	False	False

not	
Operando	Resultado
True	False
False	True

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

Valores lógicos (III)

- **Operadores de comparación**

Python proporciona ocho operadores de comparación que devuelven un resultado *booleano* True o False. Todos tienen la misma precedencia, mayor que la de los operadores lógicos

Operador	Descripción
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual que
!=	Distinto a
is [not]	Igualdad de objetos
[not] in	Pertenencia a colección

```
>>> 2 < 1
False
>>> 1 < 2
True
>>> 3 < 5 >= 2
True
>>> 1 != 0
True
>>> 2 > 4  or  5 == 5  and  2 <= -2
False
>>> (3 + 4*2 - 11) != 0
False
```

Python la evalúa como:
(3<5) and (5>=2)

Operadores

- **Tabla de operadores**

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	`**`	Binario	Por la derecha	1
Identidad	`+`	Unario	—	2
Cambio de signo	`-`	Unario	—	2
Multiplicación	`*`	Binario	Por la izquierda	3
División	`/`	Binario	Por la izquierda	3
División entera	`//`	Binario	Por la izquierda	3
Módulo (o resto)	`%`	Binario	Por la izquierda	3
Suma	`+`	Binario	Por la izquierda	4
Resta	`-`	Binario	Por la izquierda	4
Igual que	`==`	Binario	—	5
Distinto de	`!=`	Binario	—	5
Menor que	`<`	Binario	—	5
Menor o igual que	`<=`	Binario	—	5
Mayor que	`>`	Binario	—	5
Mayor o Igual que	`>=`	Binario	—	5
Negación	`not`	Unario	—	6
Conjunción	`and`	Binario	Por la izquierda	7
Disyunción	`or`	Binario	Por la izquierda	8

Variables (I)

Variables y asignaciones

- Las variables son elementos o estructuras del lenguaje que posibilitan que nuestros programas almacenen valores para su uso posterior

```
>>> pulgadas_a_metros = 0.0254
>>> 50 * pulgadas_a_metros
1,27
```

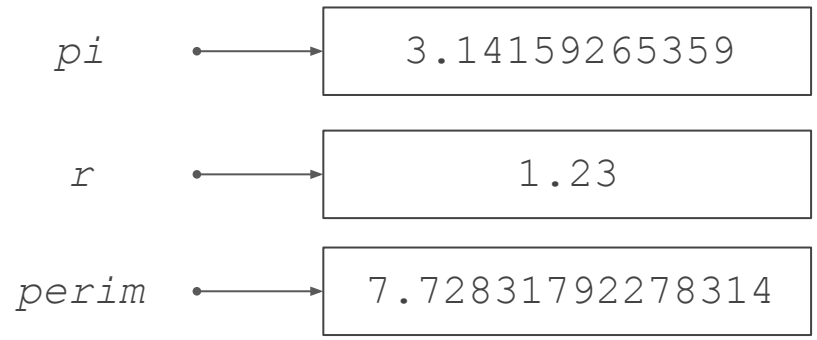
- En la primera de las líneas anteriores se ha creado una variable de nombre *pulgadas_a_metros* y se le ha dado el valor (asignación). Al asignar un valor a una variable que no existía (inicialización), Python reserva un espacio en la memoria, almacena el valor en él y crea una asociación entre el nombre de la variable y dicha dirección de memoria.

Variables (II)

- Una vez creada la variable, podremos usarla en posteriores sentencias de nuestro programa. En el momento de la ejecución, el nombre de la variable será "sustituido" por el valor que en ese momento tenga la posición de memoria a la que apunta
- Mediante el operador de asignación (=) (no confundir con el operador de comparación ==) podremos cambiar el valor de cualquier variable existente.
- De forma general, la asignación es: *variable = expresión*
  1) se evalúa la expresión a la derecha del símbolo igual (=)
  2) se guarda el valor resultante en la variable indicada a la izquierda
- La primera operación sobre una variable debe ser la asignación de un valor (inicialización). Si se intenta usar una variable no inicializada generará un error de tipo *NameError*

Variables (III)

```
>>> pi = 3.14159265359

>>> r = 1.23

>>> perim = 2 * pi * r

>>> perim
7.72831792278314

>>> r = 2

>>> perim = 2 * pi * r

>>> perim
12.56637061436
```



Creación de pi (inicialización)

Creación de r (inicialización)

Creación de perim (inicialización)

asignación de nuevo valor

asignación de nuevo valor

```
>>> perim = pi * diametro
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
NameError: name 'diametro' is not defined
```

Error: variable "diametro" no inicializada

Variables (IV)

- **Identificadores o nombre de variable**
- El nombre de una variable es su *identificador*. Para que un identificador sea válido, debe estar formado por letras, dígitos numéricos (no puede ser el primer carácter) y/o el carácter de subrayado (_)
- Un identificador no puede coincidir con una de las palabras reservadas o clave del lenguaje, es decir palabras que ya tienen un significado predefinido para Python

False	None	True	and	as	assert	break	class	continue
def	del	elif	else	except	finally	for	from	global
if	import	in	is	lambda	nonlocal	not	or	pass
raise	return	try	while	with	yield			

- Python distingue entre mayúsculas y minúsculas. Como guía de estilo se suele emplear nombres representativos en minúsculas que, cuando son de dos o más palabras, se unen mediante _

Variables (V)

- **Asignaciones con operador**
- Al igual que muchos otros lenguajes, Python soporta la forma compacta de asignación con operador asociado.
- Todos los operadores aritméticos disponen de su versión compacta con asignación. Son de la forma:

operador=

(sin espacio entre el operador y el signo =)

- Estas construcciones permiten reemplazar sentencias como:

```
contador = contador + 1
```

por otras más compactas como:

```
contador += 1
```

```
>>> a = 5
>>> b = 2
>>> a += 4 * b
>>> a
13
>>> z = 1
>>> z *= 3
>>> z **= 2
>>> z -= 1
>>> z //= 2
>>> z %= 4
>>> z
0
```

Cadenas de Caracteres (I)

El tipo String

- Las denominadas cadenas de caracteres, que son secuencias de caracteres (letras, números, espacios, símbolos,...) se emplean para la representación de información textual.
- Python dispone del tipo str para crear variables que almacenen y operen con cadenas de caracteres
- En Python, las cadenas de caracteres deben ir encerradas entre comilla simple (‘) o comilla doble (“)

```
>>> cadena = "cadena es una variable de tipo String para guardar esta cadena"
>>> cadena
'cadena es una variable de tipo String para guardar esta cadena'
>>> type(cadena)
<class ‘str’>
```

Cadenas de Caracteres (II)

- Si queremos que nuestro texto contenga comillas, podemos "*escaparlas*" precediéndolas del carácter (\) o bien, encerrar con comilla doble un texto que contenga comillas simples (y viceversa)

```
>>> texto = "Este texto lleva \"comillas\""
>>> otro_texto = 'Y éste "también"'
>>> texto, otro_texto
('Este texto lleva "comillas"', 'Y éste "también"')
```

- El "*escapado*" se utiliza también para introducir "*códigos especiales*", como el salto de línea (\n) o el tabulador (\t), para formatear la salida.

```
>>> nuevo_texto = "Una línea\ny otra"
>>> nuevo_texto
'Una linea\ny otra'
>>>print(nuevo_texto)
Una linea
y otra
```

Cadenas de Caracteres (III)

- Una característica de Python en el tratamiento de cadenas es que que permite un triple entrecomillado, con comilla simple o doble, para mantener en la impresión el mismo formato del texto introducido

```
>>> texto_multilinea = '''
Esto es un
    ejemplo de un
        texto   multilínea
'''
>>> texto_multilinea
'\nEsto es un\n    ejemplo de un\n\t\ttexto\tmultilínea\n'
>>> print(texto_multilinea)

Esto es un
    ejemplo de un
        texto   multilínea


>>>
```

Cadenas de Caracteres (IV)

- **Operaciones con cadenas de caracteres**
- Python emplea el operador suma (+) para concatenar cadenas de texto

```
>>> cadena_1 = "Hola, "
>>> cadena_2 = cadena_1 + "Mundo!"
>>> print(cadena_2)
Hola, Mundo!
```

Esto es lo que en programación denominamos sobrecarga de operadores. El operador tendrá más de un significado y el compilador (o intérprete) decide en cada momento cuál usar en función del tipo de los operandos.

```
>>> 2 + 2
4
>>> "2" + "2"
'22'
>>> "2" + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Cadenas de Caracteres (V)

- Otro operador aritmético *sobrecargado* que podemos usar en Python con cadenas es el operador producto (*).
- Este operador nos permite repetir una cadena, a modo de patrón, tantas veces como indique el valor numérico asociado

```
>>> "Hola, "*5
'Hola, Hola, Hola, Hola, Hola, '
>>> separador = "-"*60
>>> print(separador)
------------------------------------------------------------
>>> len(separador)
60
>>> print(separador + "\n" + "\t"*3 + "CABECERA\n" + separador)
------------------------------------------------------------
                        CABECERA
------------------------------------------------------------
```

- len() es una *función predefinida*, de las que luego hablaremos, que devuelve el número de caracteres de la cadena pasada como argumento

Cadenas de Caracteres (VI)

- **Métodos de cadenas de caracteres**
- Python trata internamente a las cadenas (igual que al resto de tipos primitivos) como objetos o instancias de la clase str.
  Esta clase dispone de numerosos métodos que nos permiten realizar todo tipo de operaciones con las cadenas: convertir a mayúsculas o minúsculas, encontrar un carácter, sustituciones, división en *tokens* (*split*),...
- Para invocar cualquiera de estos métodos, la sintaxis será:

objeto_cadena.método(*arg1,..., argn*)

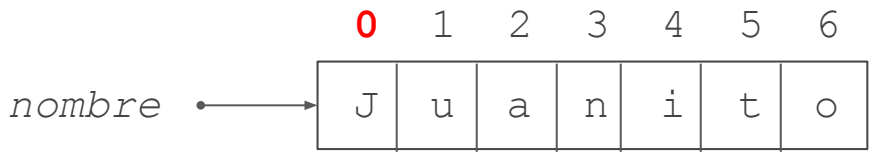
```
>>> cadena_1 = "Hola"
>>> cadena_2 = cadena_1.upper()
>>> print(cadena_2)
HOLA
>>> "Dónde está Wally?".find("Wally")
11
```

Cadenas de Caracteres (VII)

- En la URL https://docs.python.org/3.7/library/stdtypes.html#string-methods está la lista completa de métodos de la clase String (<str>). Por ejemplo:
  - upper() convierte a mayúsculas, lower() a minúsculas, title() pasa la primera letra de cada palabra a mayúsculas, capitalize() deja el primer carácter en mayúsculas y el resto en minúsculas
  - find(*subcadena*[,*ini*[,*fin*]]) devuelve la posición de la primera ocurrencia de *subcadena* entre las posiciones *ini* y *fin*
  - count(*subcadena*[,*ini*[,*fin*]]) cuenta el número de veces que aparece *subcadena* dentro de la cadena entre las posiciones *ini* y *fin*
  - split([*separador*]) crea una lista de *tokens* separando mediante el carácter *separador* (" " por defecto), splitlines() separa por líneas
  - strip(), lstrip() y rstrip() para eliminar espacios en blanco
  - replace (*old*, *new*[, *n*]) devuelve una cadena con *n* reemplazos de la subcadena *old* por *new*

Cadenas de Caracteres (y VIII)

- Las cadenas son, en realidad, secuencias de caracteres almacenados en posiciones consecutivas de memoria (*array*). Utilizando el identificador de la cadena y un índice, podemos acceder a caracteres individuales o subcadenas (pero no modificar el valor).



Las cadenas en Python son inmutables!!

```
>>> nombre = "Juanito"
>>> nombre[0]
'J'
>>> print(nombre[2])
a
>>> nombre[4:6]
'it'
>>> nombre[4:]
'ito'
>>> "Otra cadena"[:4]
'Otra'
```

El índice del primer carácter es **0**

Extraemos subcadenas con dos índices [i:j] i posición inicial, j posición final (no incluida)

Si omitimos i ([:j]) o j ([i:]), se extrae desde el principio o hasta el final

Funciones predefinidas (I)

- *abs(n)*, devuelve el valor absoluto del número *n*
- *float(exp)*, devuelve el número en punto flotante resultado de convertir la expresión *exp*

```
>>> float(5)
5.0
>>> float("5.2")
5.2
```

- *int(exp)*, devuelve el entero resultado de convertir la expresión *exp*

```
>>> int(5.8)
5
>>> int("-5")
-5
```

"trunca", no redondea

- *str(n)*, devuelve la cadena resultante de convertir el número *n*

```
>>> str(3.456)
'3.46'
```

Funciones predefinidas (y II)

- *round(n[,prec])*, devuelve el entero resultante de redondear *n*. Si se especifica *prec*, devuelve un flotante redondeado a esa precisión (*prec*)

```
>>> round(3.656)
4
>>> round(3.656, 2)
'3.66'
```

- *len(cadena)*, devuelve el número de caracteres de *cadena*
- *bin(n)*, *oct(n)*, *hex(n)*, devuelven una cadena que representa al número *n* en la base correspondiente
- *ord(char)*, devuelve el valor numérico del carácter *char*
- *chr(n)*, devuelve el carácter asociado al código numérico *n*

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

Salida formateada (I)

El método *format*

- A la hora de mostrar los resultados de la ejecución de nuestros programas, nos encontraremos con el problema de mostrarlos de la forma más clara y conveniente posible
- La clase <str> nos proporciona el método *format()* para producir salidas formateadas
- La cadena a mostrar contendrá una serie de "*campos de reemplazo*" rodeados por *llaves* { } que serán reemplazados por la lista de argumentos de *format*, de forma que {0} es el primer argumento, {1} el segundo y así sucesivamente

```
>>> a = 3.5234
>>> 'la suma de {0} + {0} + {1} es {2}'.format(a, 2.0, 2*a + 2.0)
'la suma de 3.5234 + 3.5234 + 2.0 es 9.046800000000001'
```

Salida formateada (II)

- Python nos permite indicar en el *campo de reemplazo* el formato preciso con que queremos que se visualice:

{campo:formato}

formato → [[*relleno*] *alineamiento*] [*signo*] [#] [0] [*ancho*] [.*precisión*] [*código de tipo*]

- *relleno*, carácter de relleno para los espacios de alineamiento (por defecto, espacio en blanco)
- *alineamiento*, < izquierda, > derecha (defecto), ^ centrado
- *signo*, + el signo siempre aparece, - sólo para negativos (defecto)
- #, los enteros en binario, octal y hexadecimal se preceden con prefijo
- *0*, si aparece se usa 0 para sustituir espacios en blanco
- *ancho*, número mínimo de caracteres que ocupará el valor representado
- *.precision*, número de decimales para números en punto flotante
- *código de tipo*, carácter que indica el tipo de representación

Salida formateada (III)

- **Códigos de tipo**
  - números enteros
    - *b*, binario
    - *c*, carácter Unicode
    - *d*, base diez (defecto)
    - *o*, octal
    - *x*, hexadecimal
    - *n*, como d, pero formato local
  - números en coma flotante
    - *e*, notación exponente
    - *f*, notación de punto fijo
    - *g*, notación general (defecto)
    - *n*, como d, pero formato local
    - *%*, multiplicado por 100, en formato *f* y seguido de símbolo %

Salida formateada (IV)

- **Ejemplos**

```
>>> 'El {0:>10} formateado'.format(123)
'El        123 formateado'
>>> 'El {0:0>10} formateado'.format(123)
'El 0000000123 formateado'
>>> 'El {0:@<10} formateado'.format(123)
'El 123@@@@@@@ formateado'
>>> 'El {0:b} formateado'.format(123)
'El 1111011 formateado'
>>> 'El {0:#b} formateado'.format(123)
'El 0b1111011 formateado'
>>> 'El {0:#x} formateado'.format(123)
'El 0x7b formateado'
>>> 'El {0:_^10.2f} formateado'.format(123.45678)
'El __123.46__ formateado'
>>> 'El {0:_>10.2f} formateado'.format(123.45678)
'El ____123.46 formateado'
>>> 'El {0:.4e} formateado'.format(123.45678)
'El 1.2346e+02 formateado'
```

Salida formateada (y V)

Las *f-strings*

- La versión 3.6 de Python introdujo un nuevo mecanismo para la generación de salidas formateadas, las *f-strings*
- Permiten insertar expresiones dentro de los literales *String* empleando una sintaxis mínima. Dichas expresiones se evaluarán en tiempo de ejecución, mostrando mejor rendimiento que los mecanismos anteriores
- La *f-string* irá precedidad por el carácter f y las expresiones que contenga se encerrarán entre llaves { }
- Podemos aplicar los especificadores de formato vistos anteriormente

```
>>> nom = 'Juan'
>>> edad = 25
>>> alt = 1.836
>>> f'Me llamo {nom}, mido {alt:.2f} y el próximo año tendré {edad + 1} años'
'Me llamo Juan, mido 1.84 y el próximo año tendré 26 años'
```

Entrada de datos (I)

La función *input*

- La función predefinida *input()* permite la captura de la entrada de datos del usuario por el teclado.
- Al llamar a esta función, se detiene la ejecución del programa y todo lo que vaya escribiendo el usuario se irá almacenando en un *buffer*. No se retornará el control al programa hasta que pulse la tecla *Enter*
- La función *input()* retornará todo el contenido del *buffer* de modo que podamos almacenarlo en una variable

```
>>> datos_de_entrada = input()
Hola qué tal
>>> datos_de_entrada
'Hola qué tal'
```

Entrada de datos (y II)

- La función *input()* puede tener como argumento un texto que se mostrará antes de solicitar la entrada del usuario
- Es importante tener en cuenta que el valor devuelto por la función *input()* es una cadena de texto, por lo que, dependiendo del caso, deberemos convertirla para poder operar con ella

```
>>> dato1 = input('Introduce un número: ')
Introduce un número: 2
>>> dato2 = input('Introduce otro: ')
Introduce otro: 3
>>> print('La suma de', dato1, 'y', dato2, 'es', dato1 + dato2)
La suma de 2 y 3 es 23
>>> print('Agghhh! No! la suma es', int(dato1) + int(dato2))
Agghhh! No! la suma es 5
```