

Tipos Base

enteros, reales, lógicos, textos

```
int 783 0 -192
float 9.23 0.0 -1.7e-6
bool True False
str "Uno\nDos" 'Pa\mi'
```

cadena inmutable, secuencia ordenada de letras

nueva línea
multilínea
tabulación
escaped

Tipos Contenedores

- secuencia ordenada, índices rápidos, valores repetibles
- sin orden previo, llave única, índices rápidos; llaves = tipos base o tuplas

```
list [1,5,9] ["x",11,8.9] ["texto"] []
tuple (1,5,9) 11,"y",7.4 ("texto",) ()
dict {"llave":"valor"} {}
set {"key1","key2"} {1,9,3,0} set()
```

expresión separada por comas
asociaciones llave/valor

Identificadores

para variables, funciones, módulos, clases... nombres

a..zA..Z seguidos de a..zA..Z_0..9

- acentos permitidos pero mejor evitarlos
- prohibido usar palabras de python
- discrimina minúsculas/MAYÚSCULAS

© a toto x7 y_max BigOne
© 8y and

Conversiones

type (expresión)

```
int("15") se puede especificar la base en el 2º parámetro
int(15.56) trunca la parte decimal (round(15.56) para redondear)
float("-11.24e8")
str(78.3) y la representación literal repr("Texto")
bool use comparadores (con ==, !=, <, >, ...), resultado lógico, valor de verdad
list("abc") use cada elemento de una secuencia ['a','b','c']
dict([(3,"tres"),(1,"uno")]) use cada elemento de una secuencia {1:'uno',3:'tres'}
set(["uno","dos"]) use cada elemento de una secuencia {'one','dos'}
":".join(['toto','12','pswd']) unir textos 'toto:12:pswd'
"textos y espacios".split() separar textos ['textos','y','espacios']
"1,4,8,2".split(",") separar textos ['1','4','8','2']
```

Asignación de Variables

```
x = 1.2+8+sin(0)
y,z,r = 9.2,-7.6,"bad"
```

valor o expresión calculada
nombre de variable (identificador)
nombre de variable
contenedor con varios valores (aquí una tupla)
incrementar
decrementar
«indefinido» valor constante

Índices de secuencias

para listas, tuplas, textos, ...

```
len(lst) → 6
```

acceso individual a los valores [índice]

```
lst[1] → 67 lst[0] → 11 primer valor
lst[-2] → 42 lst[-1] → 1968 último valor
```

acceso a sub-secuencias via [inicio corte: fin corte: pasos]

```
lst[1:3] → [67, "abc"]
lst[-3:-1] → [3.14, 42]
lst[:3] → [11, 67, "abc"]
lst[4:] → [42, 1968]
```

Omitiendo un parámetro de corte → de principio / hasta el fin.

En secuencias mutables, se puede eliminar elementos con del lst[3:5] y modificar asignando lst[1:4]=['hop',9]

Lógica Booleana

Comparadores: < > <= >= == !=

a and b y lógico
ambos simultáneamente o lógico

a or b uno, el otro, o ambos
no lógico

not a valor constante verdadero

True valor constante falso

Bloques de Setencias

```
sentencia madre:
├── bloque de sentencias 1...
├── ...
├── sentencia madre:
├── └── bloque de sentencias 2...
├── ...
└── sentencia siguiente a bloque 1
```

Sentencias Condicionales

bloque de sentencias que solo se ejecuta si la condición es verdadera

```
if expresión lógica:
    └── bloque de sentencias
```

puede tener varios elif, elif... y solo un else al final, ejemplo:

```
if x==42:
    # solo si la expresión lógica x==42 se cumple
    print("realmente verdad")
elif x>0:
    # si no, si la expresión lógica x>0 se cumple
    print("seamos positivos")
elif tamosListos:
    # sino, si la variable lógica tamosListos es verdadera
    print("mira, estamos listos")
else:
    # en todos los otros casos
    print("todo lo demás no fue")
```

Matemáticas

ángulos en radianes

```
from math import sin,pi...
sin(pi/4) → 0.707...
cos(2*pi/3) → -0.4999...
acos(0.5) → 1.0471...
sqrt(81) → 9.0
log(e**2) → 2.0 etc. (cf doc)
```

números reales... valores aproximados!

Operadores: + - * / // % **

(1+5.3)*2 → 12.6

abs(-3.2) → 3.2

round(3.57,1) → 3.6

bloque de sentencias que se repite mientras la condición se cumpla

Sentencia Bucle Condicional

while expresión lógica:
→ bloque de sentencias

s = 0
i = 1 } inicializaciones **antes** del bucle

condición con al menos un valor variable (aquí **i**)

```
while i <= 100:  
    # sentencias se ejecutan mientras i ≤ 100  
    s = s + i**2  
    i = i + 1
```

print("suma:", s) } resultado computado luego del bucle
; cuidado con hacer bucles infinitos!

Control de Bucles

break salir inmediatamente
continue siguiente iteración

$$S = \sum_{i=1}^{100} i^2$$

bloque de sentencias ejecutadas para cada ítem de un contenedor o iterador

Sentencia Bucle Iterador

for variable **in** secuencia:
→ bloque de sentencias

recorre los **valores** de la secuencia
s = "un texto" } inicializamos **antes** del bucle
cnt = 0
variable de bucle, valor manejado por la sentencia **for**
for c in s:
 if c == "t":
 cnt = cnt + 1
print("encontramos", cnt, "'t'")

Contamos cantidad de letras **t** en el texto

recorrer un dict/set = recorrer la secuencia de llaves
use cortes para recorrer una subsecuencia

Recorrer los **índices** de una secuencia

- modificar el ítem correspondiente al índice
- acceder ítemes alrededor del índice (antes/después)

```
lst = [11, 18, 9, 12, 23, 4, 17]  
perdidos = []  
for idx in range(len(lst)):
```

```
    val = lst[idx]  
    if val > 15:  
        perdidos.append(val)  
        lst[idx] = 15
```

Limita los valores mayores a 15, guarda los valores perdidos.

```
print("modif:", lst, "-perd:", perdidos)  
Recorrer simultáneamente los índices y valores de una secuencia:  
for idx, val in enumerate(lst):
```

Entrada / Salida

```
print("v=", 3, "cm :", x, ", ", y+4)
```

ítemes a imprimir: valores literales, variables, expresiones
parámetros de **print**:

- **sep=" "** (separador de ítemes, espacio por omisión)
- **end="\n"** (caracter final, por omisión nueva línea)
- **file=f** (escribir a archivo, por omisión salida estándar)

```
s = input("Instrucciones: ")
```

input siempre retorna un **texto**, convertir a tipo requerido (revisar Conversiones al reverso).

Operaciones sobre Contenedores

len(c) → cuenta ítemes

min(c) **max(c)** **sum(c)**

Nota: Para diccionarios y conjuntos, las operaciones son sobre las **llaves**.

sorted(c) → copia ordenada

valor in c → lógico, operador de membresía **in** (de ausencia, **not in**)

enumerate(c) → iterador sobre (índice, valor)

Especial para **contenedores de secuencias** (listas, tuplas, textos):

reversed(c) → iterador inverso

c*5 → duplicados **c+c2** → concatenar

c.index(val) → posición

c.count(val) → cuenta ocurrencias

modificar lista original

Operaciones sobre Listas

```
lst.append(item)
```

añadir ítem al final

```
lst.extend(seq)
```

añadir secuencia de ítemes al final

```
lst.insert(idx, val)
```

insertar ítem en un determinado índice

```
lst.remove(val)
```

elimina el primer ítem con determinado valor

```
lst.pop(idx)
```

elimina determinado ítem y retorna su valor

```
lst.sort() lst.reverse()
```

ordena / invierte la lista original

Operaciones en Diccionarios

```
d[llave]=valor d.clear()
```

```
d[llave]→valor del d[llave]
```

```
d.update(d2) } actualiza/añade
```

```
d.keys() } asociaciones
```

```
d.values() } ver las llaves, valores
```

```
d.items() } y asociaciones
```

```
d.pop(llave)
```

Operaciones en Conjuntos

Operadores:

| → unión (caracter barra vertical)

& → intersección

- ^ → diferencia/diferencia simétrica

< <= > >= → relaciones de inclusión

```
s.update(s2) s.add(valor)
```

```
s.remove(llave)
```

```
s.discard(llave)
```

Archivos

guardar datos a disco, volver a leerlos

```
f = open("doc.txt", "w", encoding="utf8")
```

variable para operaciones

nombre de archivo (+ruta...)

modo de apertura

- 'r' lectura
- 'w' escritura
- 'a' añadir...

codificación de caracteres en archivo:
utf8 ascii latin1 ...

escritura

```
f.write("hola")
```

text file → lee / escribe solo textos, convierte convertir al tipo requerido.

```
f.close() } no olvidar cerrar el archivo al final
```

Cerrado automático **pytónico**: **with open(...)** **as f:**

muy común: bucle iterativo para leer las líneas de un archivo de textos

```
for linea in f:
```

→ # bloque que procesa cada línea

vacía si llegamos al fin

```
s = f.read(4)
```

si se omite cuantos caracteres, se lee todo el archivo

```
s = f.readline()
```

lectura

Generador de Secuencias de Enteros

uso frecuente en bucles iterativos **for**

por omisión 0

range([inicio, fin[, paso]])

no inclusivo

```
range(5) → 0 1 2 3 4
```

```
range(3, 8) → 3 4 5 6 7
```

```
range(2, 12, 3) → 2 5 8 11
```

range retorna un « generador », convertir a lista para ver los valores, por ejemplo:
print(list(range(4)))

nombre de función (identificador)

Definir Funciones

parámetros nombrados

```
def nombfunc(p_x, p_y, p_z):
```

```
    """documentación"""
```

→ # bloque de sentencias, calcula result., etc.

```
    return res
```

valor resultado.

si no hay resultado, se retorna: **return None**

parámetros y variables sólo existen dentro del bloque y durante la llamada a la función ("caja negra")

Invocar Funciones

```
r = nombfunc(3, i+2, 2*i)
```

un argumento por parámetro

obtener el valor de retorno (opcional)

Formato de Textos

directivas de formato

valores a formatear

```
"model {} {} {}".format(x, y, r) → str
```

```
"{selección:formato!conversión}"
```

□ **Selection:**

```
2
```

```
x
```

```
0.nombre
```

```
4[llave]
```

```
0[2]
```

□ **Formating:**

relleno alineación signo anchomin.precisión~anchomax tipo

<> ^ + - espacio 0 al inicio para rellenar con 0

enteros: **b** binario, **c** character, **d** decimal (omisión), **o** octal, **x** or **X** hexa...

reales: **e** or **E** exponencial, **f** or **F** punto fijo, **g** or **G** general (omisión),

% porcentaje

cadenas: **s** ...

□ **Conversión:** **s** (texto legible) or **r** (representación literal)