

Tipos de datos dinámicos (duck typing) (I)

- El tipo de cada variable se determina de forma dinámica.
- El tipo de dato se consulta con `type()` o `isinstance()`

```
>>> x = "1234"
```

```
>>> type(x)
```

```
<type 'str'>
```

```
>>> x = 34
```

```
>>> type(x)
```

```
<type 'int'>
```

```
>>> isinstance(x, int)
```

```
True
```

© Prof. Miguel García Silvente

57

Tipos de datos dinámicos (duck typing) (II)

- Asignación de tipo dinámico débil o fuerte

- Perl (débil):

```
$b = '1.2'
```

```
$c = 5*$b; # conversión implícita
```

```
# de tipo: '1.2' -> 1.2
```

- Python (fuerte):

```
b = '1.2'
```

```
c = 5*b
```

```
# es ilegal porque no hay conversión implícita
```

© Prof. Miguel García Silvente

58

- Operaciones de E/S
- Sentencia condicional
- Bucles

Entrada estándar

- Se utiliza `input` (en versión 2.x `raw_input`)
`x = input("Num.datos: ")`
- `x` sería de tipo string, para que sea int:
`x = int(input("Número:"))`
- ¿Qué ocurre si no es un entero?

Salida estándar (I)

- Se utiliza **print**
- Se puede usar de varias formas:

```
print ('el cuadrado de', x, ' es', x * x)
```

```
print('el cuadrado de {} es {}'.format(x, x**2))
```

```
print('el cuadrado de %d es %d' % (x, x**2))
```

Salida estándar (II)

- Por defecto se separan con espacios y cada **print** va a una línea.

```
>>> print('esto','es', 'un', 'test')
```

esto es un test

- Modificadores:

- para evitar que se separen por un espacio se usa **sep**:

```
>>> print('esto','es', 'un', 'test', sep="")
```

estoesuntest

- para evitar el final de línea se usa **end**:

```
>>> print(item, " ", end=""); print ("2")
```

Salida estándar (III)

Una cadena se puede formatear con **format**

```
print('los datos son {} y {}'.format(x, x**2))
```

```
print('los datos son {1} y {0}'.format(x, x**2))
```

```
print('el cuadrado de {num1} es  
{num2}'.format(num1=x, num2=x**2))
```

Sentencia condicional: if else

La cláusula **else** es opcional:

```
if x< 0 :
```

```
    print("Negativo")
```

```
else :
```

```
    print("Positivo")
```

Sentencia condicional: if elif

Se pueden encadenar con **elif**:

```
if x < 0 :  
    print ("Negativo")  
elif x==0 :  
    print("Cero")  
else :  
    print ("Positivo")
```

© Prof.Miguel García Silvente

65

Función lambda con if else

```
>>> x = 100  
>>> resultado = (-1 if x < 0 else 1)  
>>> print (resultado)  
1  
>>> fact = lambda n : 1 if n==1 or n==0 else  
n*fact(n-1)  
>>> fact(5)  
120
```

© Prof.Miguel García Silvente

66

Selección múltiple: match (I)

Existe desde la versión 3.10

match estado :

case 200 :

 print("Correcto")

case 404 :

 print("No existe")

case 401 | 402 | 403 :

 print("No permitido")

Selección múltiple: match (II)

La coincidencia puede ser de otro forma:

match punto :

case x, y :

 print("Es un punto 2D: (" , x , " , " , y , ")")

case x, y, z :

 print("Es un punto 3D: (" , x , " , " , y , " , " , z , ")")

case _ :

 print ("Dimensión desconocida:", punto)

Selección múltiple: match (III)

```
def ordenar(sec):
    match sec:
        case [] | []:
            return sec
        case [x, y] if x <= y:
            return sec
        case [x, y]:
            return [y, x]
        case [] | []:
            case [x, y, z] if x <= y <= z:
                return sec
            case [x, y, z] if x >= y >= z:
                return [z, y, x]
            case [p, *rest]:
                a = ordenar([x for x in rest if x <= p])
                b = ordenar([x for x in rest if p < x])
                return a + [p] + b
```

© Prof. Miguel García Silvente

69

Bucles for (I)

- Se itera sobre una serie de elementos:
`for l in (12, 45, 23, 9, 12, 20) :
 print (l)`
- Se puede realizar iterando sobre una secuencia generada con **range**:
`for l in range(20) :
 print (l)
for r in range(1, 5, 2) :
 print (r)`

© Prof. Miguel García Silvente

70

Bucles for (II)

Se puede ejecutar código después de iterar:

```
for x in (1,2,3,4,5,6) :  
    print (x)  
else:  
    print ("bucle terminado")
```

```
for x in (1,2,3,4,5,6) :  
    print (x)  
    if x > 3 : break # no es aconsejable  
else:  
    print (" bucle llega hasta el final")
```

© Prof.Miguel García Silvente

71

Bucles for (III)

¿Qué ocurre en el siguiente caso?

```
for i in range(12) :  
    print(i)  
    i += 2
```

© Prof.Miguel García Silvente

72

Iterar con varias variables: zip

- Sobre pares:

```
for x,y in zip(range(12), range(100, 120)) :  
    print (x, y)
```

- O sobre tres elementos:

```
r = zip(range(20), range(12), (23, 45, 16))  
for x, y, z in r :  
    print(x,y,z)
```

Bucles while

Se ejecuta mientras se da una condición:

```
x = 1  
while x < n :  
    x = x * 2
```

Tipos de datos más complejos

- Tupla: `tuple`
- Lista: `list`
- Diccionario: `dict`
- Conjunto: `set`
- Array: `array`

Mutabilidad

- Un objeto es inmutable si una vez creado no se puede modificar:
 - No tiene funciones que permitan cambiar su contenido
- Los objetos mutables poseen funciones para modificar su contenido.

Tipos de datos secuencia

- Tuplas
- Listas
- Cadenas de caracteres

Tipos de secuencias

1. Tupla (*tuple*)

- Una secuencia **immutable** y ordenada de elementos
- Los elementos pueden ser de distinto tipo, incluyendo tipos compuestos

2. Cadena de caracteres (*str*)

- **Immutables**
- Similares a una tupla

3. Lista (*list*)

Secuencia ordenada y **mutable** de elementos de distintos tipos.

(en python 3.x existe también el tipo range)

Tuplas, listas,strings (I)

- Las tuplas se definen usando comas (y paréntesis)
`>>> tu = (23, 'abc', 4.56, (2,3), 'def')`
- Las listas usando corchetes y separando con comas
`>>> li = ["abc", 34, 4.34, 23]`
- Las cadenas usando comillas (" , ' o """).
`>>> st = "Hello World"`
`>>> st = 'Hello World'`
`>>> st = """Esta es una cadena con`
`varias líneas que usa comillas triples.""""`

© Prof.Miguel García Silvente

79

Tuplas, listas,strings (II)

- Acceso a un elemento mediante corchetes.
- El primer elemento es el 0

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]    # segundo item de la tupla.
'abc'
```

```
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]    # Segundo item de la lista.
34
```

```
>>> st = "Hello World"
>>> st[1]  # Segundo carácter del string.
'e'
```

© Prof.Miguel García Silvente

80

Índices positivos y negativos

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Índice **positivo**: desde la izquierda, empezando en 0

```
>>> t[1]
```

'abc'

Índice **negativo**: contando por la derecha, empezando con -1

```
>>> t[-3]
```

4.56

Trocear (I)

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Devuelve una copia de un contenedor compuesto por un subconjunto de los miembros. Empieza en el primer índice y termina antes del segundo:

```
>>> t[1:4]
```

('abc', 4.56, (2,3))

Se pueden usar índices negativos:

```
>>> t[1:-1]
```

('abc', 4.56, (2,3))

Trocear (II)

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Si se omite el primer índice se empieza en el primer elemento:

```
>>> t[:2]
```

```
(23, 'abc')
```

Si se omite el segundo se llega hasta el último elemento:

```
>>> t[2:]
```

```
(4.56, (2,3), 'def')
```

Copiar la secuencia completa

[:] hace una copia de la secuencia completa

```
>>> t[:]
```

```
(23, 'abc', 4.56, (2,3), 'def')
```

Dos ejemplos para objetos mutables:

```
>>> l2 = l1 # l2 es una referencia a l1,
```

```
    # si se cambia uno, afecta al otro
```

```
>>> l2 = l1[:] # copias independientes,
```

```
    # 2 referencias
```

Valores nulos

- Listas:

```
>>> t = []
```

- Cadenas de caracteres:

```
>>> cad = ""
```

- Tuplas:

```
>>> t = ()
```

El operador 'in'

- Comprueba si un valor aparece en un contenedor:

```
>>> t = [1, 2, 4, 5]
```

```
>>> 3 in t
```

False

```
>>> 4 in t
```

True

```
>>> 4 not in t
```

False

- En las cadenas, comprueba subcadenas

```
>>> a = 'abcde'
```

```
>>> 'c' in a
```

True

```
>>> 'cd' in a
```

True

```
>>> 'ac' in a
```

False

El operador +

El operador + produce una nueva tupla, lista o cadena concatenando los argumentos

```
>>> (1, 2, 3) + (4, 5, 6)  
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]  
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"  
'Hello World'
```

© Prof.Miguel García Silvente

87

El operador *

Produce una nueva tupla, lista o cadena repitiendo el contenedor original.

```
>>> (1, 2, 3) * 3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> "Hello" * 3  
'HelloHelloHello'
```

© Prof.Miguel García Silvente

88

Las listas son mutables

```
>>> li = ['abc', 23, 4.34, 23]
```

```
>>> li[1] = 45
```

```
>>> li
```

```
['abc', 45, 4.34, 23]
```

- Se pueden cambiar directamente.
- `li` tiene la misma referencia de memoria cuando terminamos.

Las tuplas son inmutables

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

```
>>> t[2] = 3.14
```

Traceback (most recent call last):

File "<pyshell#75>", line 1, in -toplevel-
tu[2] = 3.14

TypeError: object doesn't support item assignment

- No es posible cambiar una tupla.
- **La inmutabilidad permite que sean más rápidas que las listas.**

Inmutabilidad vs mutabilidad

¿Qué ocurre en este caso?

```
>>> t = (1, 2, [3, 4])
```

```
>>> t[2][1] = 5
```

```
>>> t
```

```
(1, 2, [3, 5])
```

Mutabilidad y paso de parámetros

¿Qué ocurre en este caso?

```
def f(v, i) :
```

```
    v[i] += 1
```

```
>>> aux = [1,2,3,4]
```

```
>>> f(aux, 2)
```

```
>>> aux
```

```
[1, 2, 4, 4]
```

Operaciones sobre listas

```
>>> li = [1, 11, 3, 4, 5]
```

```
>>> li.append('a')
```

```
>>> li
```

```
[1, 11, 3, 4, 5, 'a']
```

```
>>> li.insert(2, 'i') # en la posición 2 habrá una i
```

```
>>> li
```

```
[1, 11, 'i', 3, 4, 5, 'a']
```

© Prof.Miguel García Silvente

93

+, extend, append

- `+` crea una lista nueva, con una nueva referencia a memoria
- `extend` opera directamente sobre la lista `li`.

```
>>> li.extend([9, 8, 7])
```

```
>>> li
```

```
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- Confusiones habituales:

- `extend` añade todos los elementos de la lista.
- `append` incluye un único elemento.

```
>>> li.append([10, 11, 12])
```

```
>>> li
```

```
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

© Prof.Miguel García Silvente

94

Operaciones sobre listas (I)

Algunos ejemplos: `index`, `count`, `remove`, `reverse`, `sort`

```
>>> li = ['a', 'b', 'c', 'b']
```

```
>>> li.index('b') # índice de la 1a aparición, si no existe  
se genera una excepción
```

1

```
>>> li.count('b') # número de veces que aparece
```

2

```
>>> li.remove('b') # elimina la 1a aparición
```

```
>>> li
```

```
['a', 'c', 'b']
```

© Prof.Miguel García Silvente

95

Operaciones sobre listas (II)

```
>>> li = [5, 2, 6, 8]
```

```
>>> li.reverse() # invierte el orden de los elementos de la lista
```

```
>>> li
```

```
[8, 6, 2, 5]
```

```
>>> li.sort()      # ordena la lista
```

```
>>> li
```

```
[2, 5, 6, 8]
```

```
>>> li.sort(key = <función>)
```

ordena la lista usando una comparación

definida por el usuario

```
cads = ['holá','12','a','abc']
```

```
print (cads)
```

```
cads.sort(key = lambda x:
```

```
len(x))
```

```
print (cads)
```

© Prof.Miguel García Silvente

96

Listas: inserción y borrado

```
>>> l = [1, 2, 3, 4, 5]
```

Borrar elementos

```
>>> l[2:3] = []
```

```
>>> del l[-1]
```

Insertar

```
>>> l[2:2] = [3]
```

```
>>> l[:0] = [7]
```

Detalles del tipo tupla (l)

- La coma es el operador de creación de una tupla (no lo es el paréntesis)

```
>>> 1,
```

```
(1,)
```

- Python usa paréntesis por claridad

```
>>> (1,)
```

```
(1,)
```

Detalles del tipo tupla (II)

- Es imprescindible el uso de la coma

```
>>> (1)
```

1

- Existen las tuplas vacías

```
>>> ()
```

()

```
>>> tuple()
```

()

Devolución de valores en funciones (I)

Se pueden devolver varios datos

```
def f(x) :
```

```
    return x, x*x, x*x*x # o return (x,x*x,x*x*x)
```

```
aux = f(7) # aux es de tipo tuple
```

```
a, b, c = f(5)
```

```
a, b = f(12) # Error
```

Devolución de valores en funciones (II)

También se puede hacer usando una lista:

```
def f(x) :  
    return [x, x*2, x**2]  
aux = f(7) # aux es de tipo list
```

```
a, b, c = f(5)  
a, b = f(12) # Error
```

Resumen: Tuplas vs. Listas

- Las listas son más lentas pero más potentes:
 - las listas se pueden modificar,
 - tienen multitud de operaciones y métodos,
- las tuplas son inmutables y tienen menos características.
- Se puede convertir de un tipo al otro con las funciones `list()` y `tuple()`:
`li = list(tu)`
`tu = tuple(li)`

Uso de listas (I)

Usando índices:

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

```
suma = 0
```

```
for i in range(len(L)) :
```

```
    suma += L[i]
```

Uso de listas (II)

Usando la filosofía de python

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

```
suma = 0
```

```
for i in L :
```

```
    suma += i
```

Uso de listas (III)

Usando índices:

```
L = [1, 2, 3, 4, 5]  
L_pares = []  
for i in range(len(L)) :  
    if L[i] % 2 == 0 :  
        L_pares.append(L[i])
```

Uso de listas (IV)

Usando la filosofía de python:

```
L_pares = []  
for i in L :  
    if i % 2 == 0 :  
        L_pares.append(i)
```