

# Python para Análisis de datos: Introducción

## Sesión 2

Miguel Expósito Martín (exposito\_m@cantabria.es)

2 Abril 2019

variables, estructuras de datos y de control, expresiones, sentencias y funciones

variables

expresiones

ejecución condicional

funciones

bucles

strings

lists

dicts

tuples & files

variables, estructuras de datos y de control, expresiones,  
sentencias y funciones

variables

¿qué es una constante?

un valor fijo, que nunca cambia

```
print(123)
print(15.2)
print('curso python')
pi = 3.14
print(pi)
```

recordad: ¡¡ojo con las palabras reservadas!!

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

## ¿qué es una variable?

una variable es una posición de memoria a la que se asocia un nombre o etiqueta  
se les asigna valores con el operador =

```
x = 12.2
print(x)
y = 'variable'
x = 100
print(x)
```

## nomenclatura

- ▶ deben comenzar por una letra o guion bajo \_
- ▶ pueden contener letras, números y guiones bajos
- ▶ son sensibles a mayúsculas/minúsculas

## declaraciones

```
x = 1 # declaración de asignación  
x = x + 1 # asignación de expresión  
print(x) # declaración print
```

¿cuáles son las variables, operadores, constantes, funciones?

## declaraciones de asignación

una declaración de asignación consiste en una expresión a la derecha del igual y una variable a la izquierda donde almacenar el resultado

```
x = 3.9 * x * ( 1 - x )
```

## expresiones

### expresiones numéricas

operador	operación
+	suma
-	resta
	multiplicación
/	división
	potenciación
%	resto

## ejemplos

```
x = 2
x = x + 2
print(x)
y = 430 * 11
print(y)
z = y / 1000
print(z)
j = 26
k = j % 5
print(k)
```

## precedencia de operadores

1. paréntesis
2. potenciación
3. multiplicación, división y resto
4. suma y resta
5. de izquierda a derecha

```
x = 1 + 2 ** 3 / 4 * ( 5 + 4 )
```

## tipos

toda variable en Python tiene un tipo

Python sabe la diferencia entre `number` y `string`

```
x = 5 + 8
print(x)
type(x)
greeting = 'hola' + ' ' + 'Miguel'
print(greeting)
type(greeting)
```

con cadenas de texto, + concatena

## tipos

algunas operaciones están prohibidas...

```
greeting = 'hola' + ' ' + 'Miguel'
greeting = greeting + 1
```

¡no se puede sumar 1 a una cadena de texto!



## tipos numéricos

- ▶ enteros
- ▶ coma flotante

```
x = 1
type(x)
y = 48.9
type(y)
z = 1.0
type(z)
```

## conversiones de tipo

en una expresión, int se convierte implícitamente a float

conversiones explícitas: `int()` y `float()`

```
print(float(24) + 50)
i = 35
type(i)
f = float(i)
print(f)
type(f)
```

## conversiones de tipo

muy utilizada en logging:

```
msg = 'Número total de ocurrencias: '  
count = 2  
print(msg + str(count))
```

también se puede hacer lo contrario:

```
value= '555'  
print(int(value) + 5)
```

## división entera

la división entera produce un resultado en punto flotante

```
print(10 / 2)
```

## entrada por teclado

```
input() devuelve un string

name = input('¿Cómo te llamas?')
print('Bienvenido', name)

age = input('Edad: ')
age_to_retirement = 65 - int(age)
print('Te quedan', age_to_retirement, 'años para jubilarte')
```

## comentarios

Python ignora cualquier cosa después de una almohadilla (#)

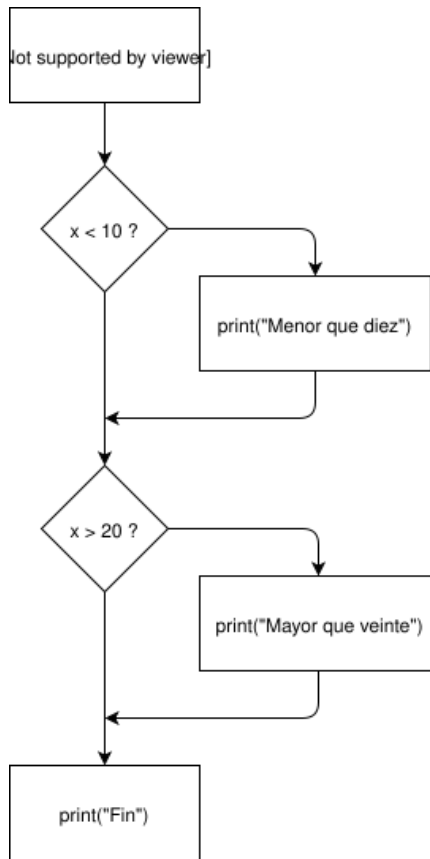
Los comentarios sirven para:

- ▶ describir lo que va a pasar en el código
- ▶ documentar quién escribió esa parte del código
- ▶ deshabilitar temporalmente una línea de código (\*)

## ejercicio

*escribir un programa que obtenga del usuario el número de horas trabajadas y el coste por hora y que calcule la paga de un empleado*

ejecución condicional



```
x = 7
if x < 10:
    print('Menor que diez')
if x > 20:
    print('Mayor que veinte')
print('Fin')
```

## operadores de comparación

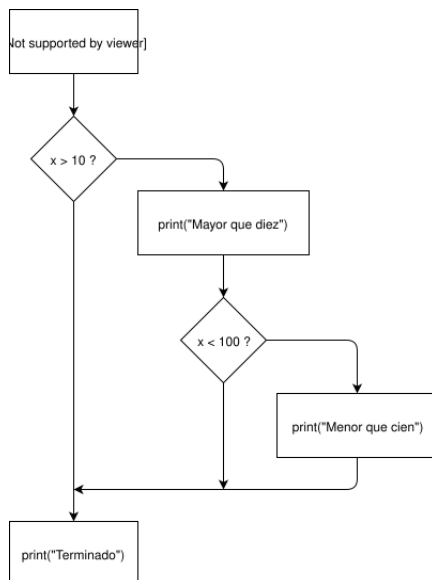
operador	operación
<	menor que
<=	menor o igual que
==	igual a
>=	mayor o igual que
>	mayor que
!=	distinto de

las expresiones que utilizan estas operaciones se evalúan a los valores True/False  
estos operadores no cambian el valor de las variables

## sangría

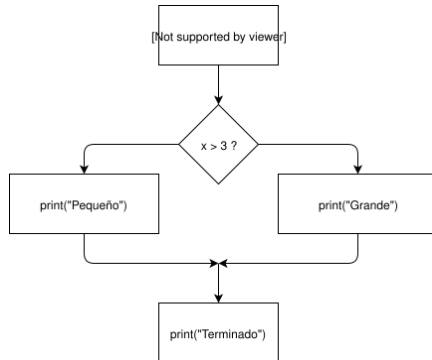
- ▶ después de un if o un for, incrementar (después de :)
- ▶ mantener para indicar el ámbito del bloque (líneas afectadas por if/for)
- ▶ reducir al nivel del padre para indicar fin del bloque
- ▶ las líneas en blanco no afectan a la sangría
- ▶ los comentarios no afectan a la sangría

## decisiones anidadas



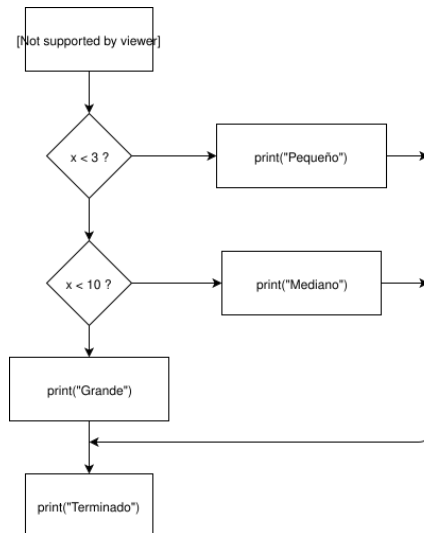
```
x = 30
if x > 10 :
    print('Mayor que diez')
    if x < 100 :
        print('Menor que cien')
print('Terminado')
```

## decisiones con dos caminos



```
x = 45
if x > 3 :
    print('Grande')
else:
    print('Pequeño')
print('Terminado')
```

## decisiones multicamino



```
x = 5
if x < 3 :
    print('Pequeño')
elif x < 10 :
    print('Mediano')
else:
    print('Grande')
print('Terminado')
```

## ojo con liarse

```
if x < 2 :  
    print('Menor que 2')  
elif x >= 2 :  
    print('Mayor que 2')  
else :  
    print('Otra cosa')
```

## estructuras try / except

- ▶ se rodea una sección de código “peligrosa” con un try / except
- ▶ si el código en el try funciona, el except se ignora
- ▶ si el código en el try falla, se salta a la sección except



## ejemplo

```
astr = 'Hola Miguel'
try:
    istr = int(astr)
except:
    istr = -1
print('Primero', istr)

astr = '123'
try:
    istr = int(astr)
except:
    istr = -1
print('Segundo', istr)
```

## ejercicio

*Reescribir el programa del cálculo de paga para pagarle todas las horas de más a partir de 40 a 1.5 veces el coste de la hora normal*

Ej: 45 horas y 10 €/hora: 475 €

## funciones

### definición

trozos de código reutilizable que toman argumentos de entrada, realizan algún tipo de cálculo y devuelven resultados

- ▶ se definen con `def` e indentando su contenido o cuerpo
- ▶ se invocan usando su nombre, paréntesis y argumentos

## ejemplo

```
def my_function():  
    print('Hola')  
    print('Adios')  
  
my_function()  
print('Pepe')  
my_function()
```

## tipos

- ▶ built-in: parte de Python (print(), input(), int())
- ▶ definidas por el usuario

¡no usar nombre de función como nombre de variable!

## argumentos

- ▶ son valores que se pasan como entrada a la función en su llamada
- ▶ permiten dirigir a la función para que lleve a cabo distintos tipos de trabajo

```
print('Pepe')
```

## parámetros

- ▶ son variables utilizadas en la definición de la función
- ▶ permiten a la función acceder a sus argumentos para una llamada particular

```
def say_hello(lang):  
    if lang == 'es' :  
        print('Hola')  
    elif lang == 'fr' :  
        print('Salut')  
    else:  
        print('Hello')  
say_hello('fr')
```

## retorno

una función retorna un resultado con return

```
def say_hello(person):  
    return "Hello " + str(person)  
  
print(say_hello('Ana'))
```

## múltiples parámetros

- ▶ simplemente, se añaden
- ▶ ¡ojo con el número y orden de parámetros!

```
def add_three(a, b, c):  
    sum = a + b + c  
    return sum  
x = add_three(3, 5, 7)  
print(x)
```

## algunos consejos

- ▶ organizar el código
- ▶ DRY!
- ▶ divide y vencerás
- ▶ hacer bibliotecas con lo que más se usa

## ejercicio

Escribir una función que, dadas las coordenadas de un punto en un plano  $(x,y)$ , imprima su cuadrante.

## bucles

### qué son

- ▶ los bucles permiten repetir la ejecución de un determinado paso
- ▶ contienen variables de iteración que cambian cada vez
- ▶ las variables de iteración suelen recorrer una secuencia de números

## ejemplo while (bucles indefinidos)

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Despegue!')
print(n)
```

¿qué hace esto?

```
n = 5
while n > 0 :
    print(n)
print('Despegue!')
print(n)
```



¿y esto?

```
n = 0
while n > 0 :
    print(n)
print('Despegue!')
print(n)
```

salir de un bucle

```
break
while True :
    text= input('Introduce q para terminar: ')
    if text == 'q' :
        break
    print(text)
print('Hecho')
```

## salir de una iteración

```
continue
while True :
    text = input('Introduce q para terminar, c para otra oportunidad: ')
    if text == 'q' :
        break
    elif text == 'c':
        continue
    print(text)
print('Hecho')
```

## bucles definidos

- ▶ necesidad de recorrer una lista de items
- ▶ se ejecutan un número exacto de veces
- ▶ iteran sobre los miembros de un conjunto

```
for i in [5, 4, 3, 2, 1] :
    print(i)
print('Despegue!')
```

también se pueden iterar strings

```
it_crowd = ['Izu', 'Saro', 'Seco']  
for it_guy in it_crowd :  
    print('Pregúntaselo a: ', it_guy)  
print('Hecho!')
```

de forma general

1. inicializar las variables
2. encontrar algo que hacer para cada item de forma separada, actualizando la variable
3. ver el resultado

## ejercicio

*cuál es el máximo de la siguiente lista: 9, 41, 12, 3, 74, 15*

## conteos

```
numeros = [9, 41, 12, 3, 74, 15]
total = 0
for numero in numeros :
    total = total + 1
print('Hecho! el total es:', total)
```

## media aritmética

```
numeros = [9, 41, 12, 3, 74, 15]
total = 0
suma = 0
for numero in numeros :
    total = total + 1
    suma = suma + numero
print('Hecho! la media es:', suma / total)
```

## filtrado

```
numeros = [9, 41, 12, 3, 74, 15]
for numero in numeros :
    if numero > 20 :
        print('Mayor que 20')
print('Hecho! )
```

## is / is not

- ▶ utilizados en expresiones lógicas
- ▶ significa “es lo mismo que”
- ▶ similar a ==, pero más fuerte
- ▶ is devuelve True si dos variables apuntan al mismo objeto
- ▶ == devuelve True si los objetos a los que apuntan dos variables son iguales

```
x = None
if x is None :
    print('No hay valor para x')
```

## strings

## más sobre strings

- ▶ un string es una secuencia de caracteres
- ▶ se pueden usar comillas simples o dobles
- ▶ aunque contenga números, un string es un string
- ▶ el operador + significa concatenación
- ▶ se comparan con ==
- ▶ se pueden convertir strings a números con `int()` o `float()`
- ▶ en Python 3, todos los strings son unicode
- ▶ son **inmutables** (no se pueden modificar)

## indexado y rebanado

indexado: como un vector que empieza en 0

```
fruit = 'plátano'
print fruit[3]
print(len(fruit)) # longitud de la cadena
print fruit[0:5] # slicing
print fruit[1:]  # slicing hasta el final
```

## iteración

```
fruit = 'plátano'
for letra in fruit:
    print(letra)
```

## in como operador lógico

```
fruit = 'plátano'
'n' in fruit
```



## string library

- ▶ funciones por defecto
- ▶ se invocan añadiéndolas a la variable

**importante: no modifican el string original, sino que devuelven uno modificado**

## ejemplos

```
find() busca la primera ocurrencia de un substring  
fruit = plátano  
pos = fruit.find('ta')  
print(pos)
```

```
mayúsculas y minúsculas  
greeting = 'Hola Miguel'  
mins = greeting.lower()  
print(mins)  
mays = greeting.upper()  
print(mays)
```

## ejemplos

```
buscar reemplazar                                quitar espacios en blanco
greeting = 'Hola Miguel'                        (stripping)
another_greeting = greeting.replace('Miguel', 'Pepe')
print(another_greeting)                         greeting = '  Hola Miguel  '
                                                greeting.lstrip()
                                                greeting.rstrip()
                                                greeting.strip()
```

## ejemplos

```
prefijos
greeting = 'Que tenga un buen día'
greeting.startswith('Que')
```

## ejercicio

Extraer el nombre del servidor de la siguiente cabecera de correo electrónico:

From miguel.exposito@apps.cantabria.es Sat Jan 5 09:14:16 2008

lists

## definición

```
[5, 89, 'Miguel']
```

- ▶ colecciones de elementos
- ▶ un elemento puede ser cualquier tipo de objeto
- ▶ una lista puede estar vacía
- ▶ son mutables: se puede cambiar un elemento usando el operador índice
- ▶ son **ordenadas**

## indexado, rebanado y longitud

```
my_list = [5, 12, 21, 40, 68]
print(my_list[1]) # indexado
my_list[1] = 7 # mutabilidad
print(my_list)
print(my_list[2:3]) # rebanado
print(len(my_list)) # longitud
```

## rangos

devuelven una my\_list de números en el intervalo [0,N-1]

```
print(range(4))
```

## concatenación

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
```

## ordenación

```
a = [7, 4, 9, 1, 8]
print(a.sort())
```

## añadir y quitar elementos

```
my_list = list()
my_list.append('Miguel')
my_list.append('Paco')
my_list.append('Alberto')

pop quita el último elemento y lo devuelve

print(my_list.pop())
print(my_list)

también se puede especificar qué elemento (índice)
```

## otras funciones

`len, max, min, sum`

`split` rompe un string en una lista de strings

```
my_list = 'lista de cuatro palabras'
```

```
palabras = my_list.split()
```

```
print(palabras)
```

se puede especificar un delimitador

## ejercicio

From miguel.exposito@apps.cantabria.es Sat Jan 5 09:14:16 2008

## dicts

### definición

- ▶ también son colecciones de elementos
- ▶ permiten realizar operaciones similares a las de las bases de datos
- ▶ también llamados arrays asociativos, son la colección más potente en Python
- ▶ **no son ordenados**
- ▶ sus elementos no se indexan con un número, sino con una etiqueta de búsqueda



## ejemplo

```
ages = dict()
ages['miguel'] = 37
ages['pepito'] = 88
ages['carmen'] = 25
print(ages)

ages['pepito'] = ages['pepito'] + 2
print(ages)
```

## lists vs dicts

```
my_list = list()
my_list.append(21)
my_list.append(183)
print(my_list)
my_list[0] = 23
print(my_list)
```

```
my_dict = dict()
my_dict['edad'] = 21
my_dict['curso'] = 183
print(my_dict)
my_dict['edad'] = 23
print(my_dict)
```

## diccionarios con literales

```
shopping_list = {'patatas': 3, 'york': 100, 'yogur': 6}
```

se puede crear un diccionario vacío así:

```
empty_dict = {}
```

## uso común: múltiples contadores

```
names = {}  
names['Paco'] = 1  
names['Pepe'] = 1  
print(names)  
names['Paco'] = names['Paco'] + 1  
print(names)
```

## cuidado cuando la clave no existe

si se utiliza una clave que no existe, aparecerá un error

Solución: `get()`

```
counts = {}  
counts['mesa'] = counts.get('mesa', 0) + 1  
counts['silla'] = counts.get('silla', 0) + 1  
print(counts)  
counts['mesa'] = counts.get('mesa', 0) + 1  
print(counts)
```

## iteración en diccionarios

```
counts = {'mesa': 2, 'silla': 4, 'reposapiés': 3}  
for key in counts:  
    print(key, counts[key])  
  
counts = {'mesa': 2, 'silla': 4, 'reposapiés': 3}  
for key, value in counts.items():  
    print(key, value)
```

## obtener claves y valores

```
counts = {'mesa': 2, 'silla': 4, 'reposapiés': 3}
print(counts.keys())
print(counts.values())
print(counts.items())
```

## contar palabras

- ▶ hacer split del texto en palabras
- ▶ iterar a lo largo de las palabras
- ▶ usar un diccionario para contar cada palabra de forma independiente

## ejercicio

Contar el número de ocurrencias de cada palabra en la siguiente frase:

*Tanto monta, monta tanto, Isabel como Fernando.*

OJO! Comas y mayúsculas...

## ejercicio avanzado

Obtener una lista con los nombres de proveedor ordenados por prioridad.

**Sugerencia:** usar funciones anónimas (lambda)

```
providers = {
  'places': {
    'priority': 1,
    'eligible': False
  },
  'google': {
    'priority': 2,
    'eligible': True
  },
  'bing': {
    'priority': 4,
    'eligible': False
  },
  'mapbox': {
    'priority': 3,
    'eligible': True
  }
}
```

## tuples & files

### tuples

un tipo de secuencia similar a una lista... pero inmutable

```
x = [9, 8, 7]  
x[2] = 6  
print(x)
```

```
z = (5, 4, 3)  
z[2] = 0
```

¿qué no se puede hacer?

- ▶ sort
- ▶ append
- ▶ reverse

lists vs tuples

- ▶ las tuplas son más eficientes (memoria y rendimiento)
- ▶ se pueden usar como un diccionario sin claves
- ▶ se suelen usar dentro de listas
- ▶ son comparables elemento a elemento

**no se pueden añadir ni eliminar elementos**

## asignación

```
(x,y) = (4, 'Pepe')  
print(y)
```

## ficheros

- ▶ un fichero de texto puede verse como una secuencia de líneas
- ▶ para abrirse, se crea un apuntador para manipularlo

```
my_file = open('prueba.txt')  
for line in my_file:  
    print(line.rstrip())
```