

Programación en Python

Máster en Bioinformática Aplicada a Medicina Personalizada y Salud



Curso 2019/2020

```
bicycles.py x motorcycles.py x hrs_worked.py
1 hrs = int(input("Enter Hours:"))
2 rph = float(input("Enter Rate per Hour:"))
3
4 if hrs <= 40 :
5     total_pay = hrs * rph
6     print(total_pay)
7 else :
8     ot_pay = ((hrs - 40) * (1.5 * rph))
9     base_pay = 40 * rph
10    total_pay = base_pay + ot_pay
11    print(total_pay)
```

Carlos González Calvo (carlosgo@ucm.es)

Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Índice

- Creación de código en Python
- Literales, variables y operadores
- Entrada y salida de datos
- Listas
- Cadenas
- Bloques
- Sentencias condicionales
- Bucles
- Diccionarios
- Ficheros
- Funciones
- Expresiones regulares

Creación de código

- Un programa es una secuencia de órdenes que la electrónica del computador sabe interpretar.
- La interpretación de estas órdenes es secuencial.
- El computador sólo entiende niveles de voltaje (0s y 1s).
- Lenguajes de programación:
 - Evitan tener que escribir los programas mediante secuencias de 0s y 1s.
 - Implican la existencia de compiladores y enlazadores.
 - Todos suelen tener unos componentes básicos comunes.

Creación de código

- Todo lenguaje de programación viene descrito por:
 - Un **léxico**. Conjunto de palabras que pueden usarse en un programa:
 - Palabras clave.
 - Nombres de variables, constantes, subprogramas, etc.
 - Una **sintaxis**. Formas válidas para mezclar los elementos léxicos.
 - Una **semántica**. Significado de las expresiones definidas con una sintaxis correcta.

Creación de código

Planteamiento del problema

Diseño del algoritmo

Representación de los datos a utilizar

} Sobre papel

Codificación del programa

Compilación

Depuración

} Sobre computador

Creación de código: Ejemplo

– Planteamiento del problema

- Realizar un programa que pida al usuario que introduzca la nota de todos los alumnos de una clase y calcule la media. Si la media está por debajo de 4 el programa indicará por pantalla que los resultados son malos, si está por encima de 7 indicará que los resultados son buenos.

– Diseño del algoritmo (pseudo-código)

Versión 1

```
Leer nota alumno 1
Guardar nota alumno 1
Acumular nota
Leer nota alumno 2
Guardar nota alumno 2
Acumular nota
...
Leer nota alumno n
Guardar nota alumno n
Acumular nota
Calcular media
Si media < 4 imprimir mensaje negativo
Si media > 7 imprimir mensaje positivo
```

Versión 2

```
Para cada alumno i {
    Leer nota alumno i
    Guardar nota alumno i
    Acumular nota
}
Calcular media
Si media < 4 imprimir mensaje negativo
Si media > 7 imprimir mensaje positivo
```

Creación de código: Ejemplo

- Representación de los datos a utilizar
 - Definición de datos como variables o constantes.
 - Tamaño y tipo de datos.
 - Influye sobre la cantidad de memoria utilizada.
 - Datos estructurados.
 - Vectores. Conjunto de datos relacionados mediante índices.
 - Ejemplo:
 - Vector que almacena todas las notas: nota_alumno[i] debe almacenar números reales positivos.
 - Variable que almacena la media: media debe almacenar números reales positivos.

Creación de código: Ejemplo

- Codificación
 - Supone elegir un lenguaje de programación (Python)

```
# Ejemplo: Calcula la media de las notas de 40 alumnos

total= 0;
nota_alumno=[None]*40

for i in range(0, 40):
    nota_alumno[i] = float(input("Introduce la nota del alumno: \n"))
    total= total + nota_alumno[i];

media = total/ 40;
print("media =", media, "\n")

if (media < 4):
    print("¡Los resultados del examen han sido malos!\n")
if (media > 7):
    print("¡Los resultados del examen han sido buenos !\n")
```


Creación de código: Ejemplo

- Compilación y enlazado
 - La compilación es el proceso de traducir las órdenes escritas en un lenguaje de alto nivel a lenguaje entendible por la máquina.
 - Las órdenes más habituales del lenguaje máquina son muy sencillas:
 - Movimientos de datos entre una posición de memoria y un registro.
 - Suma, resta, multiplicación o división de datos en registros.
 - Desplazamientos.
 - Comprobaciones de bifurcación:
 - Si el contenido de un registro es igual a un valor entonces bifurca.
 - Saltos incondicionales.
 - El enlazado permite unir programas compilados independientemente determinando las posiciones relativas de los datos en memoria.

Desarrollo y ejecución de un programa

- Depurado

- Posibles errores:

- Al compilar pueden aparecer errores de sintaxis. Hay que corregirlos antes de poder ejecutar el programa.
 - Al ejecutar pueden presentarse errores porque el programa no funcione como se esperaba:
 - Uso de depuradores para buscar el error. Permiten:
 - » La ejecución instrucción a instrucción.
 - » Añadir *breakpoints* (puntos de parada de la ejecución) en el programa.
 - » Visualizar el contenido de una variable en un determinado momento de la ejecución.

Crear y ejecutar programas en Python

- Crear un programa:
 - Debe llevar la extensión .py
- Ejecutar un programa:
 - Permisos de ejecución

`python pepe.py`

ejecuta el programa

Literales, variables y operadores

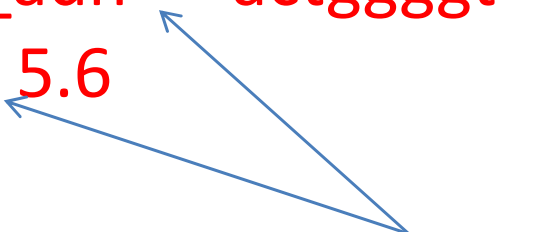
Literales, variables y operadores

- **Tipos básicos de datos:**
 - **Constantes (literales):** Mantienen siempre su valor.
 - Enteros (int): **12**
 - Decimales (float) **34.5**
 - Cadenas de texto (str): **“esto es una cadena”**
 - Booleanos (bool): **True False**
 - **Variables:** Una variable es un nombre que referencia a un valor. Ese valor puede variar a lo largo del programa. Para dar un valor a una variable usamos la **sentencia de asignación:**

Entre comillas



cadena_adn = “actggggt”
media = 5.6



Asignación

Literales, variables y operadores

- **Nombres de variables:**

- Pueden tener cualquier longitud
- Están compuestas por letras, números y _
- No pueden comenzar con un número
- Deberían usarse nombres de variables con información útil pero no excesivamente largos
- Hay una serie de palabras claves que no pueden usarse como nombres de variable porque ya tienen significado:

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

Literales, variables y operadores

- **Operadores:** Manipulan variables y constantes

- Operadores sobre números: **+** **-** ***** **/** **%** ******
- Operadores de cadenas:
 - concatenar **+**
 - duplicar *****
- Operadores lógicos:
 - y-lógica **and**
 - o-lógica **or**
 - not-lógica **not**
- Operadores de comparación: **<** **>** **<=** **>=** **==** **!=**

Adn="actttg"**+**"cccttt"

Módulo

Exponenciación

Ej:

media = total/40

adn2 = adn1+"acctt"

Entrada y salida de datos

Entrada de datos: Función `input`

- `x = input("Dame un dato:")`
 - Guarda en la variable `x` la **cadena** tecleada por el usuario, tras aparecer en pantalla Dame un dato. El programa se queda esperando hasta que se introduce un dato.
 - Si queremos convertirlo en un dato entero o en un real, debemos hacerlo explícitamente con la función `int` o `float`.
 - `x = int(input("Dame un dato:"))`
 - `x = float(input("Dame un dato:"))`
 - `x = str(3.1415)` # convierte el numero 3.1415 en una cadena

Comentario

Salida de datos

- Función `print`('cadena de caracteres', 'otra cadena')
- Imprime las cadenas de caracteres que aparecen separadas por comas o el valor de las variables

```
media = 5.54
```

```
print("El valor de la media es: ", media)
```

Por pantalla aparece:

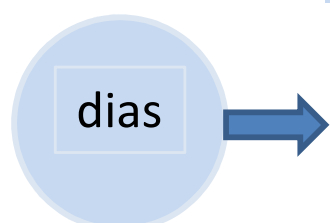
```
El valor de la media es: 5.54
```

Listas

Listas

- Lista: Es una colección de datos estructurada numerando sus elementos. Cada elemento puede ser de un tipo diferente.
- Se define separando sus elementos con comas y colocándolos entre []
- Ejemplo:

`dias = ['lunes', 'martes', 'miercoles', 'jueves', 'viernes', 'sabado', 'domingo']`



A light blue circle containing the word 'dias' in a white box. A blue arrow points from this circle to the first column of the table below.

Número de orden	Valor
0	lunes
1	martes
2	miercoles
3	jueves
4	viernes
5	sabado
6	domingo

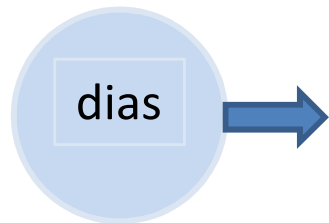
Listas

- **Acceso a un elemento:**

- `X = dias[3]` el número de orden comienza en 0

- **Modificación de un elemento:**

- `dias[2] = "mercredi"`

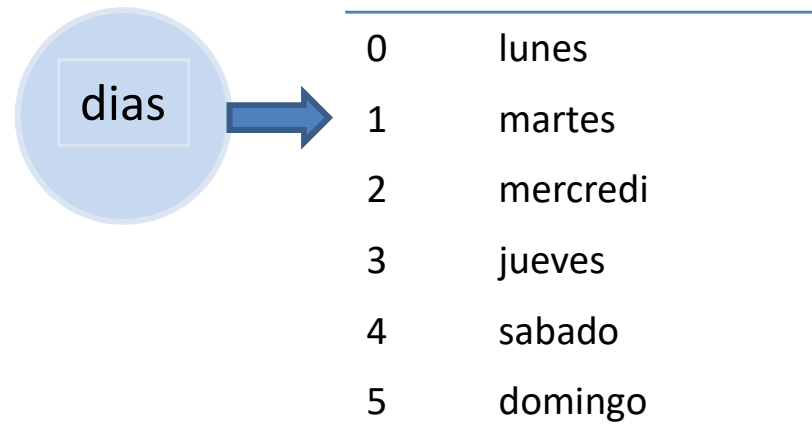


0	lunes
1	martes
2	mercredi
3	jueves
4	viernes
5	sabado
6	domingo

Listas

- **Borrar un elemento:**

- **del dias[4]**



Operaciones sobre listas

- **Troceado:** Se usa para acceder a rangos de elementos en vez de a elementos individuales.
- Devuelve otra lista con los elementos que van desde el primer índice al último menos 1.

```
numeros=[1,2,4,5,87,56]  
trozo=numeros[2:5]  
print(trozo)
```



[4, 5, 87]

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
numeros[4:7] = [23, 45, 54]  
print(numeros)
```

```
numeros[1:3] = []  
print(numeros)
```



[1, 2, 3, 4, 23, 45, 54, 8, 9, 10]

[1, 4, 23, 45, 54, 8, 9, 10]

Lista vacía

Operaciones sobre listas

- **Suma:** **+** concatena dos listas

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
vocales = ['a', 'e', 'i', 'o', 'u']  
todo=numeros+vocales  
print(todo)
```

→ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'a', 'e', 'i', 'o', 'u']

Multiplicacion: *****, repite la lista tantas veces como se indique

```
numeros = [1, 2, 3]  
numeros=numeros*2  
print(numeros)
```

→ [1, 2, 3, 1, 2, 3]

Operaciones sobre listas

- Listas vacías: `[]`
- Reserva de espacio para una lista: `None`
`seq = [None]*10` crea una lista con 10 elementos vacíos
- Operador `in`. Comprueba si un valor forma parte de la lista, devolviendo `True` o `False`.

```
numeros = [1, 2, 3]  
resultado = 2 in numeros  
print(resultado)  
resultado = 7 in numeros  
print(resultado)
```



`True`
`False`

len: Función que devuelve el número de elementos de una lista

```
numeros = [1, 2, 3, 7]  
l=len(numeros)  
print(l)
```



`4`

Operaciones sobre listas

- **max**: Función que devuelve el elemento de mayor valor de una lista.
- **min**: Función que devuelve el elemento de menor valor de una lista.
- **list**: crea una lista a partir de una cadena.

```
numeros=[1,2,4,5,87,56]  
maximo=max(numeros)  
print(maximo)
```

```
minimo=min(numeros)  
print(minimo)
```

```
cadena=list('hola')  
print(cadena)
```



87

1

['h', 'o', 'l', 'a']

Métodos sobre listas

- Un método es una función fuertemente acoplada a un objeto.
- Sintaxis:
 - objeto.metodo(argumentos)
- Método **append**: añade un objeto al final de una lista

```
numeros=[1,2,4,5,87,56]  
numeros.append(8)  
print(numeros)
```



```
[1, 2, 4, 5, 87, 56, 8]
```

Métodos sobre listas

- Método **count**: cuenta las ocurrencias de un elemento en una lista.

```
numeros=[1,2,4,2,2,56]  
t=numeros.count(2)  
print(t)
```



3

- Método **extend**: permite añadir varios valores al mismo tiempo al final de una lista suministrando la lista que se quiere añadir.

```
numeros=[1,2,4,2,2,56]  
l = [3,56]  
numeros.extend(l)  
print(numeros)
```



[1, 2, 4, 2, 2, 56, 3, 56]

Métodos sobre listas

- Método **index**: muestra el índice de la primera aparición de un elemento. Produce una excepción si no lo encuentra.

```
numeros=[1,2,4,2,2,56]  
ind = numeros.index(56)  
print(ind)
```



5

- Método **insert**: permite insertar un elemento en una lista en una posición determinada.

```
numeros=[1,2,4,2,2,56]  
numeros.insert(3,8)  
print(numeros)
```



[1, 2, 4, 8, 2, 2, 56]

Posición de
inserción

Elemento para
insertar

Métodos sobre listas

- Método **pop**: elimina un elemento de la lista. Por defecto el último o sino el número de orden indicado. Además devuelve el valor del elemento eliminado.

```
numeros=[1,2,4,2,2,56]  
eliminado=numeros.pop(3)  
print(numeros)  
print(eliminado)
```



```
[1, 2, 4, 2, 56]  
2
```

Métodos sobre listas

- Método **remove**: elimina la primera aparición de un valor. Produce una excepción si no lo encuentra.

```
numeros=[1,2,4,2,2,56]  
numeros.remove(4)  
print(numeros)  
numeros.remove(8)  
print(numeros)
```



```
[1, 2, 2, 2, 56]
```

```
Traceback (most recent call last):
```

```
File "C:/DOCENCIA/Master bioinformatica/Master  
bioinformatica 2018-2019/Ejemplos/listas1.py", line 65, in  
<module>
```

```
    numeros.remove(8)
```

```
ValueError: list.remove(x): x not in list
```

Métodos sobre listas

- Método **sort**: ordena los elementos de una lista. La lista original no se mantiene.

```
numeros=[1,2,4,2,2,56]  
numeros.sort()  
print(numeros)
```



[1, 2, 2, 2, 4, 56]

- Método **reverse**: Da la vuelta a los elementos de una lista.

```
numeros=[1,2,4,2,2,56]  
numeros.reverse()  
print(numeros)
```



[56, 2, 2, 4, 2, 1]

Listas

Ejercicio 1:

1. Crear una lista con el nombre de 6 aminoácidos
 2. Insertar uno más al final
 3. Pedir el nombre de uno de ellos al usuario y eliminarlo
 4. Ordenar la lista alfabéticamente
 5. Imprimir el número de elementos de la lista
- Imprimir la lista después de cada paso

Ejercicio 2:

Ordenar una lista manteniendo una copia de la versión original

Cadenas



Cadenas

- Una cadena es una secuencia de caracteres.
- Ciertos caracteres especiales van precedidos de \
 - `\n` salto de linea
 - `\t` tabulador
- La mayoría de las acciones que se pueden realizar sobre listas se pueden realizar sobre cadenas: troceado, indexación, multiplicación, longitud, etc.
- Las cadenas son inmutables no se pueden modificar.

```
cad= "hola"  
cad[1]='P'
```



TypeError: 'str' object does not support item assignment

Cadenas

- **Raw strings:** si se desea que todos los caracteres que aparezcan en la cadena se consideren como los caracteres que son y no como caracteres especiales se puede definir una cadena del siguiente modo:

`x = r'c:\nopath'`

Cadenas

- Operador **in**
 - Operador booleano que toma dos cadenas y devuelve True si la primera está contenida en la segunda.

```
res='a' in 'banana'  
print(res)
```



True

Métodos sobre cadenas

- Método **find**. Busca una subcadena en una cadena. Devuelve el índice del elemento primero de la subcadena encontrada. Si no lo encuentra devuelve -1

```
frase = 'En un lugar de la Mancha'  
pos= frase.find('lugar')  
print(pos)
```

→ 6

- Método **join**. Sirve para unir los elementos de una secuencia. Estos elementos tienen que ser cadenas

```
secuencia = ['1', '2', '3']  
conector = '-'  
c=conector.join(secuencia)  
print(c)
```

→ 1-2-3

Métodos sobre cadenas

- Método **split**. Trocea una cadena en una lista usando como carácter de separación el parámetro dado:

```
frase = 'En un lugar de la mancha'  
palabras = frase.split(' ')  
print(palabras)
```

Espacio en blanco



['En', 'un', 'lugar', 'de', 'la', 'mancha']

- Método **strip**. Elimina los caracteres blancos antes y después de la cadena pero no los intermedios.

Bloques

Bloques

- Un bloque es un grupo de sentencias que se ejecutan si una condición se cumple. Puede formar parte de una sentencia condicional o de un bucle.
- Se crea un bloque al indentar las sentencias que se quiere formen parte del mismo.
- Los bloques comienzan con **:** y finalizan cuando la siguiente sentencia tiene menor indentación:

Esto es una sentencia

Esto es otra:

esta es la primera línea del bloque

esta la segunda

Esta ya no pertenece al bloque

Sentencias condicionales



Sentencias condicionales

- Valores booleanos:
 - Falso: False, None, 0, "", (), [], {}
 - Verdadero: True, Todo lo demás
 - Operadores booleanos: **and or not**
 - Ejemplo: **x<0 and x<10**

Operadores de comparación

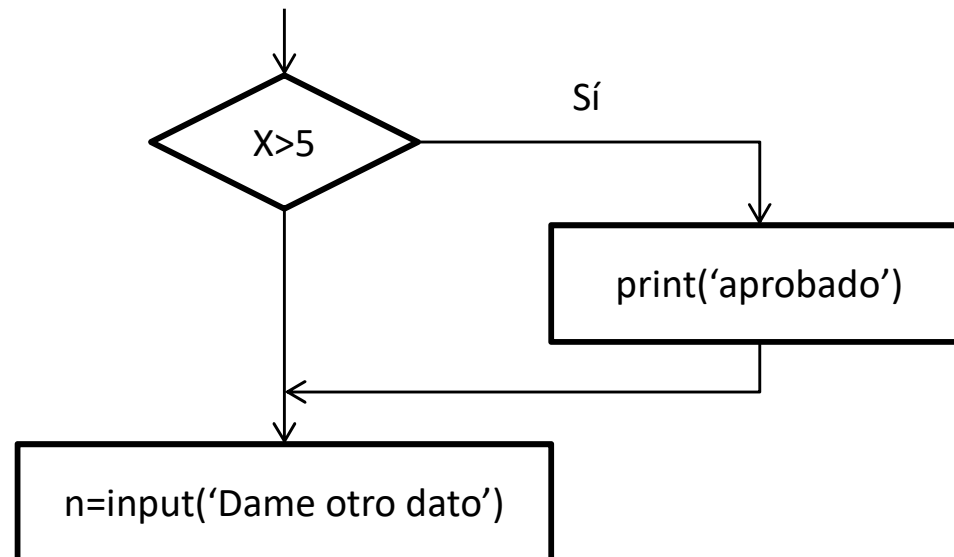
Expresión	Descripción
x == y	Igual
x < y	Menor que
x > y	Mayor que
x <= y	Menor o igual
x >= y	Mayor o igual
x != y	Distinto que
x is y	x e y son el mismo objeto
x is not y	x e y son diferentes objetos
x in y	x es un miembro de y
x not in y	x no es un miembro de y

Sentencias condicionales

- Permiten evaluar condiciones y modificar el comportamiento del programa en función del resultado de la evaluación.
- Sentencia **if**: Si se cumple la condición se ejecuta el bloque

if condición:
Bloque

```
if x>5:  
    print('aprobado')  
n=input('Dame otro dato')
```



Sentencias condicionales

- Sentencia **if-else**:

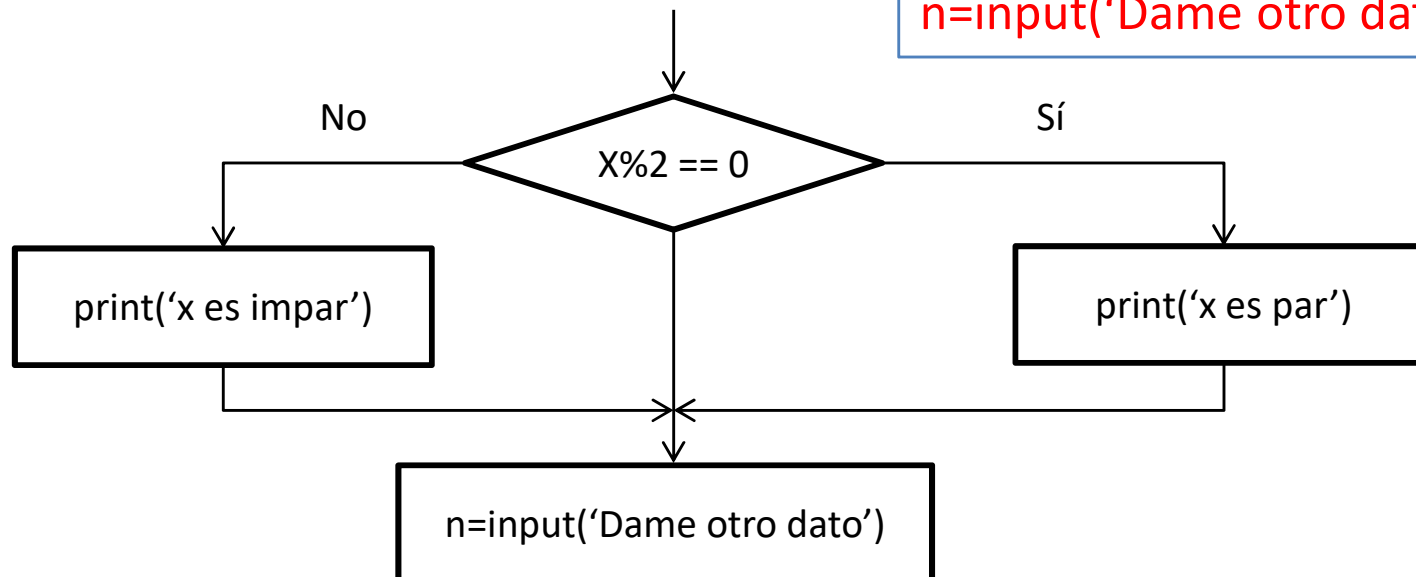
if condición:

Bloque1

else:

Bloque2

```
if x%2 == 0:  
    print('x es par')  
else:  
    print('x es impar')  
n=input('Dame otro dato')
```



Sentencias condicionales

- Sentencia **if-elif-else**

if condición1:

Bloque1

elif condición2:

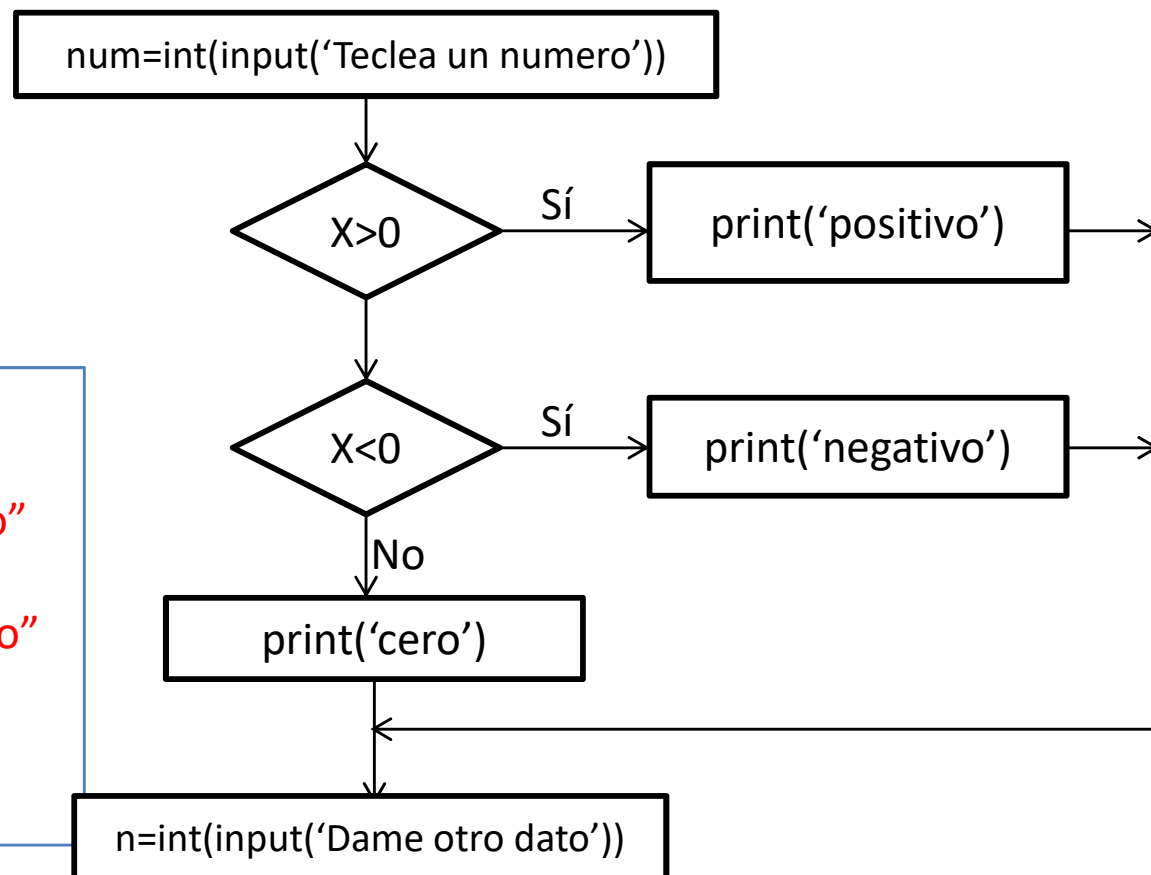
Bloque2

...

else:

Bloque3

```
num = int(input('Teclea un numero'))
if num > 0:
    print "El numero es positivo"
elif num < 0:
    print "El numero es negativo"
else:
    print "El numero es cero"
n=int(input('Dame otro dato'))
```



Bucles



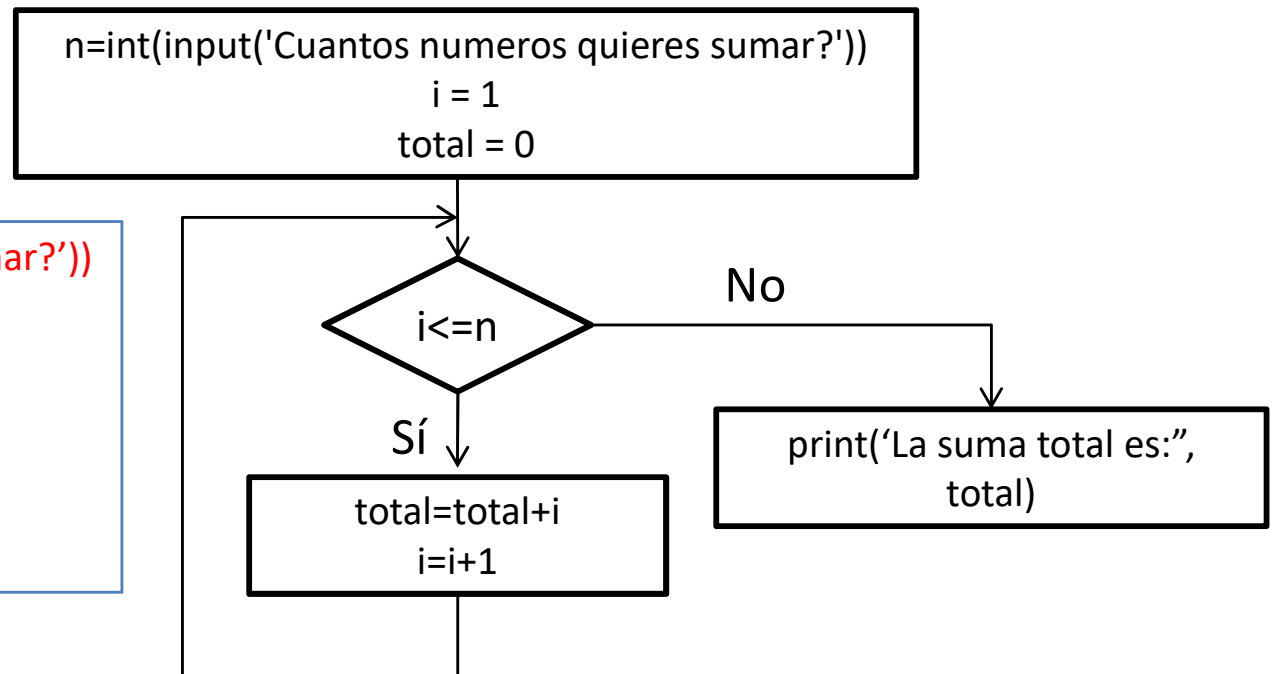
Bucles

- Permiten repetir una determinada tarea hasta que se cumpla una condición
- Bucles **while**:

while condición:

Bloque

```
n=int(input('Cuantos numeros quieres sumar?'))  
i = 1  
total = 0  
while i <= n:  
    total= total+i  
    i= i+1  
print("La suma total es:", total)
```



Bucles

- Bucles **for**:
 - Iteran sobre un listado de elementos (números, palabras, líneas, etc)

***for** nombre_variable **in** secuencia_de_valores:*
bloque

```
friends = ['Joseph', 'Glenn', 'Sally']  
for friend in friends:  
    print('Happy New Year:', friend)  
print('Done!')
```



Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!

Bucles

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```



Total: 154

```
mayor= 0
print('Before:', mayor)
for itervar in [3, 41, 12, 9, 74, 15]:
    if mayor is 0 or itervar > mayor:
        mayor= itervar
    print('Loop:', itervar, mayor)
print('MAYOR:', mayor)
```



Before: 0
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
MAYOR: 74

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print('Hay: ', count, 'letras a')
```

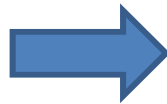


Hay: 3 letras a

Bucles

- **break:**
 - Sale del bucle.
 - Si está dentro de bucles anidados sale del más interno.
- **continue:**
 - finaliza la iteración actual de un bucle.

```
while True:  
    line = input('> ')  
    if line == '#':  
        continue  
    elif line == 'done':  
        break  
    print(line)  
print('Done!')
```



```
> hola  
hola  
> #  
> done  
Done!
```

Bucles

- `range(x)`: devuelve una lista cuyos elementos son números enteros consecutivos desde 0 hasta $x-1$.
- `range(x, y)`: devuelve una lista cuyos elementos son números enteros consecutivos desde x hasta $y-1$.
- `range(x, y, step)`: devuelve una lista cuyos elementos son números enteros desde x hasta $y-1$ partiendo de x , sumándole *step* hasta llegar a $y-1$

```
for i in range(0,5,2):  
    print(i)
```



```
0  
2  
4
```

Sentencias condicionales y bucles

Ejercicio 3:

Escribir un programa que pida al usuario números enteros hasta que introduzca un 0. Al finalizar el programa debe mostrar la cantidad de números introducidos.

Ejercicio 4:

Pedir al usuario una cadena de ADN y contar cuántas veces aparece cada una de las bases.

Hacer que esto se repita en un bucle infinito que vaya pidiendo distintas cadenas. El programa finalizará si se teclea la palabra fin.

Ejercicio 5:

Mostrar por pantalla las tablas de multiplicar del 1 al 9.

Deberá aparecer algo como:

1*1 es 1

1*2 es 2

1*3 es 3

...

Diccionarios

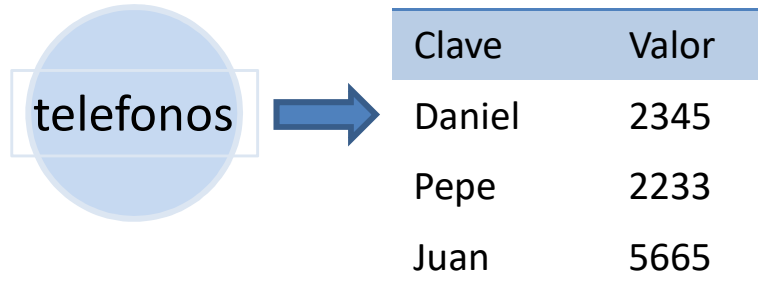


Diccionarios

- Son listas a las que se accede por una clave en vez de por un número de orden. Consisten en pares clave-valor.
- Se definen poniendo entre `{}` la lista separada por `,` de elementos clave:valor separados por `:`
 - `telefonos = {'Daniel' : 2345, 'Pepe' : 2233, 'Juan' : 5665}`
- Las claves son de cualquier tipo inmutable.
- Los valores pueden ser de cualquier tipo, incluido otro diccionario.
- No existe un orden interno de los elementos de un diccionario.
- Diccionario vacío:
 - `nombres = {}`
 - `nombres = dict()`

Diccionarios

```
telefonos = {'Daniel' : 2345, 'Pepe' : 2233, 'Juan' : 5665}
```

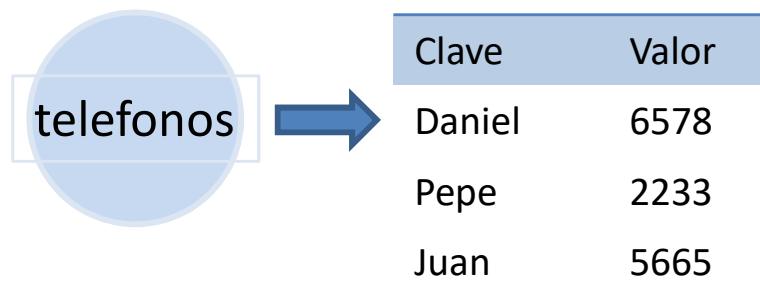


– Acceso a un elemento:

- `dato = telefonos['Daniel']` accede al valor asociado a la clave 'Daniel'

– Creación/Modificación de un elemento:

- `telefonos['Daniel'] = 6578`



Operaciones básicas sobre diccionarios

- **len(nombre)** devuelve el número de elementos del diccionario nombre.
- **del telefono['Daniel']** elimina el elemento con clave 'Daniel'.
- **k in d** comprueba si existe un elemento en el diccionario d con la clave k, devuelve True o False.

Bucles en diccionarios

- Iteración en diccionarios:
 - *Devuelve las claves*

```
d = {'x': 1, 'y': 2, 'z': 3}
```

```
for key in d:
```

```
    print(key, 'corresponde a', d[key])
```



z corresponde a 3

x corresponde a 1

y corresponde a 2



No hay orden en los diccionarios

Métodos sobre diccionarios

- **.clear()** elimina todos los elementos de un diccionario.
- **.items()** devuelve una relación de elementos en la forma (clave, valor) utilizable dentro de un for.

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}  
for k, v in knights.items():  
    print(k, v)
```



robin the brave
gallahad the pure

- **.keys()** devuelve una relación de claves, utilizable dentro de un for.

Métodos sobre diccionarios

- `.pop('clave')` devuelve el valor de 'clave' y borra el elemento
- `.update(otro_dicc)` actualiza un diccionario con los elementos de otro, si existe la clave sustituye el valor y si no existe lo crea.
- `.values()` devuelve una relación con los valores del diccionario, utilizable dentro de un for.

Diccionarios

- Diccionarios de diccionarios

```
people = {  
    'Alice': {'phone': '2341', 'addr': 'Foo drive 23'},  
    'Beth': {'phone': '9102', 'addr': 'Bar street 42'},  
    'Cecil': {'phone': '3158', 'addr': 'Baz avenue 90'}  
}
```

Ejercicios de diccionarios

Ejercicio 6:

1. Crear un diccionario con el nombre de un animal como clave, y su nombre en latín como valor.
2. Todos los datos se irán pidiendo al usuario por pantalla.
3. Cada vez que se introduzcan los datos de un animal se imprimirán todos los datos del diccionario.
4. Si un animal ya se ha introducido se le dirá al usuario.

Ejercicio 7:

1. Crear un diccionario con el nombre de las 4 bases nitrogenadas del adn como clave y como valor otro diccionario con dos elementos: la primera clave es 'abreviatura' y la segunda 'tipo' (indicando si es púrica o pirimídica).
2. Acceder al diccionario para imprimir las abreviaturas.
3. Insertar el uracilo.
4. Volver a acceder al diccionario para imprimir las abreviaturas.
5. Imprimir los nombres de todas las bases, abreviatura y tipo con algún tipo de formato

Ficheros



Ficheros

- Abrir ficheros:

`open(nombre[, modo[, buffering]])`

- El único parámetro obligatorio es el nombre del fichero a abrir.
- Esta función devuelve un objeto de tipo *file*.
- Modos:
 - 'r' lectura (opción por defecto)
 - 'w' escritura
 - 'a' añadir
 - 'b' binario (añadido a otro modo)
 - 'r+' lectura/escritura
- Ejemplo:
 - `f = open(r'c:\text\pepe.txt')` abre el fichero para lectura

Métodos sobre ficheros

- Escribir

- `f.write('Hello')` escribe la cadena Hello en el fichero

- Leer

- `f.read()` lee el resto del fichero f en una cadena
- `f.read(7)` lee los siguientes 7 caracteres del fichero

- Mover

- `f.seek(desplazamiento[, desde_donde])` mueve la posición del puntero de fichero la cantidad dada por desplazamiento.
 - El parámetro `desde_donde` puede valer:
 - 0. Es la opción por defecto e indica desplazamiento desde el principio.
 - 1. desplazamiento desde la posición actual. Podría ser negativo
 - 2. desplazamiento desde el final del fichero

- Posición

- `f.tell()` da la posición actual

Métodos sobre ficheros

- Lectura de líneas:

- `f.readline()` devuelve la línea entera
- `f.readline(n)` devuelve n caracteres de la línea
- `f.readlines` devuelve todas las líneas de un fichero como una lista

- Cierre de ficheros:

- `f.close()` cierra un fichero

Iterador de ficheros

- Iteradores de ficheros:
 - Permiten analizar línea a línea sin tener que leerlo entero en una lista.

```
f = open('nombre_fichero')
for line in f:
    procesa(line)
f.close()
```

Ejercicios de Ficheros

Ejercicio 8:

Realizar un programa que pida al usuario el nombre de un fichero, lo abra, cuente cuántas palabras de cada tipo existen y lo muestre por pantalla.

Ejercicio 9:

Abrir un fichero de genbank con datos de adn.

Contar cuántas veces aparece cada una de las bases en la cadena de DNA.

Usando un diccionario que tiene como clave los codones y como valor el aminoácido correspondiente, convertir todo el DNA en aminoácidos y guardarlo en un fichero llamado amino.txt.

Contar cuántos aminoácidos de cada tipo hay.

Por pantalla solicitar al usuario un porcentaje y mostrar cuáles son los aminoácidos por encima de ese porcentaje.

Funciones



Funciones

- Las funciones encapsulan un fragmento de código dándole un nombre, permiten pasarle parámetros y pueden devolver resultados.
- Se pueden llamar desde diferentes posiciones del programa creado (reuso de código).
- Existen muchas funciones predefinidas. Varias de estas ya las hemos utilizado.
- Podemos definir nuestras propias funciones.

Funciones predefinidas

- Funciones de **conversión de tipos**:
 - **int**. Función que toma un valor y lo convierte en entero, o da un error si no puede hacerlo.
 - **float**. Convierte enteros o cadenas a números en punto flotante (decimales)
 - **str**. Convierte su argumento a una cadena

Funciones predefinidas

- Funciones **matemáticas**:
 - Python tiene un módulo `math` que contiene muchas funciones matemáticas habituales.
 - Para usarlo hay que cargarlo previamente mediante: **`import math`**
 - Para usar cualquiera de estas funciones se debe poner previamente el módulo del que proviene, por ejemplo **`math.sin(0.7)`**
 - Ejemplos de funciones:
 - `cos(x)`, `sin(x)`, `tan(x)`, `sqrt(x)`, `log(x[, base])`, `log10(x)`, `pow(x, y)`, `factorial(x)`, `fabs(x)`

Funciones

- Definición:

```
def nombre(parametros):
```

```
    Bloque
```

- El nombre de la función sigue los mismos criterios que los nombres de variables.
- La función definida por nosotros se llama igual que las funciones predefinidas en Python, invocando su nombre.

Funciones

- El bloque puede incluir una sentencia **return()** con el valor que devuelve la función. Si no se pone nada en el return, la función devuelve None.
- La sentencia return finaliza la ejecución de la función. Si no hay return la función acaba con el bloque.

Funciones

- Parámetros de la función:
 - Parámetros formales: Los que se escriben en la definición.
 - Parámetros actuales: Los que se proporcionan cuando se llama a la función.

Funciones: Recursión

- Caso base conocido
- Caso para n en función de $n-1$

Ejercicio 10:

Escribir un programa que pida al usuario el número del que se quiere calcular el factorial:

- a) Definir una función que calcule el factorial del número.
- b) Definir una función que calcule el factorial del número de manera recursiva.

Ejercicios de funciones

Ejercicio 11:

Definir una función que convierta codones en aminoácidos

Ejercicio 12:

Definir una función que calcule la media de una lista de valores numéricos.

Ejercicio 13:

Definir una función que multiplique dos matrices de 4×4

Redefinir la función para que pueda multiplicar matrices de cualquier tamaño

Hacer un programa que pida al usuario el tamaño de las matrices a multiplicar, recoja por teclado los datos de las dos matrices y las multiplique

Hacer un programa que pida al usuario el nombre de dos ficheros donde se encuentran guardadas sendas matrices, las lea y las multiplique guardando el resultado en un fichero cuyo nombre se ha pedido al usuario

Módulos



Módulos

- Permiten encapsular en un fichero fragmentos de código que podrá ser importado por otros módulos posteriormente.
- Cualquier fichero con extensión.py funciona como módulo.

Módulos

- ¿Cómo importar un módulo?

import nombre_fichero

- Carga el fichero *nombre_fichero* en el fichero actual, lo ejecuta, y crea un objeto módulo para ser usado. Todas las variables y funciones que están en *nombre_fichero* son ahora accesibles en la forma:

nombre_fichero.variable

nombre_fichero.funcion()

Módulos

- ¿Cómo importar parte de un módulo?

from nombre_fichero **import** lista_de_nombres

- carga todos los nombres que aparecen en *lista_de_nombres* desde el fichero *nombre_fichero*.
- las variables y funciones de *lista_de_nombres* pueden usarse directamente sin poner delante *nombre_fichero*.

from nombre_fichero **import** *

- Importa todos los nombres de nombre_fichero

Módulos

- Ventajas de los módulos:
 - Reuso de código
 - Particionamiento del espacio de nombres
 - Implementar servicios o datos compartidos

Standard library: built-in functions

<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>char()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>

Módulos

- Text Processing Services
 - 1. string — Common string operations
 - 2. re — Regular expression operations
 - 3. difflib — Helpers for computing deltas
 - 4. textwrap — Text wrapping and filling
 - 5. unicodedata — Unicode Database
 - 6. stringprep — Internet String Preparation
 - 7. readline — GNU readline interface
 - 8. rlcompleter — Completion function for GNU readline
- Binary Data Services
 - 1. struct — Interpret bytes as packed binary data
 - 2. codecs — Codec registry and base classes
- Data Types
 - 1. datetime — Basic date and time types
 - 2. calendar — General calendar-related functions
 - 3. collections — Container datatypes
 - 4. collections.abc — Abstract Base Classes for Containers
 - 5. heapq — Heap queue algorithm
 - 6. bisect — Array bisection algorithm
 - 7. array — Efficient arrays of numeric values
 - 8. weakref — Weak references
 - 9. types — Dynamic type creation and names for built-in types
 - 10. copy — Shallow and deep copy operations
 - 11. pprint — Data pretty printer
 - 12. reprlib — Alternate repr() implementation
 - 13. enum — Support for enumerations

Módulos

- 9. Numeric and Mathematical Modules
 - 1. numbers — Numeric abstract base classes
 - 2. math — Mathematical functions
 - 3. cmath — Mathematical functions for complex numbers
 - 4. decimal — Decimal fixed point and floating point arithmetic
 - 5. fractions — Rational numbers
 - 6. random — Generate pseudo-random numbers
 - 7. statistics — Mathematical statistics functions
- 10. Functional Programming Modules
 - 1. itertools — Functions creating iterators for efficient looping
 - 2. functools — Higher-order functions and operations on callable objects
 - 3. operator — Standard operators as functions
- 11. File and Directory Access
 - 1. pathlib — Object-oriented filesystem paths
 - 2. os.path — Common pathname manipulations
 - 3. fileinput — Iterate over lines from multiple input streams
 - 4. stat — Interpreting stat() results
 - 5. filecmp — File and Directory Comparisons
 - 6. tempfile — Generate temporary files and directories
 - 7. glob — Unix style pathname pattern expansion
 - 8. fnmatch — Unix filename pattern matching
 - 9. linecache — Random access to text lines
 - 10. shutil — High-level file operations
 - 11. macpath — Mac OS 9 path manipulation functions

Expresiones regulares

Expresiones regulares

- Índice:
 - Expresiones regulares
 - Funciones del módulo re
 - Conjuntos de caracteres
 - Grupos
 - Alternancia
 - Repeticiones
 - Anclajes
 - Objetos MatchObject y grupos

Expresiones regulares

- Una expresión regular, ER, es una cadena que describe un patrón.
 - Su utilidad es:
 - Buscar una cadena en otra
 - Extraer las partes deseadas de una cadena
 - Buscar y reemplazar unas cadenas por otras dentro de cadenas.
 - Se basan en:
 - **Repetición**
 - **Concatenación**
 - **Alternancia**

Expresiones regulares

- La ER más sencilla es simplemente una **cadena de caracteres** que describe ese patrón
 - ‘Hola’ es la ER que describe al patrón Hola
- Lo interesante es que una única expresión regular puede describir a muchos patrones
- Por ejemplo, el punto en una ER describe a cualquier carácter
 - ‘.ython’ describe al patrón ‘python’, a ‘sython’ , o cualquier palabra de 6 letras acabada en ‘ython’

Expresiones regulares

- Caracteres especiales. Hay una serie de caracteres que tienen significado especial dentro de una ER, por ejemplo el punto.
- Como en las cadenas de caracteres \ sirve para mostrar caracteres especiales, si se quiere usar \ para dar significado a algo en ER debe ponerse doble \\
 - Los caracteres especiales son:
`{ } [] () ^ $. | * + ? \`
- Raw string:
 - Para evitar poner tantos \ se puede usar la notación de raw strings `r"texto"` donde lo que hay en texto se entiende como cada carácter individual.

Funciones del módulo **re**

- **import re**
- Funciones del Módulo re
- Para llamar a una función del módulo:

Función	Descripción
<code>compile(patron[, flags])</code>	Crea un objeto ER a partir de un patrón
<code>search(patron, cadena[, flags])</code>	Busca el patrón en la cadena
<code>match(patron, cadena[, flags])</code>	Identifica el patrón al principio de la cadena
<code>split(patron, cadena[, maxsplit=0])</code>	Trocea la cadena por las apariciones del patrón
<code>findall(patron, cadena[, flags])</code>	Devuelve una lista de todas las apariciones de un patrón
<code>sub(patron, repl, cadena[, count=0])</code>	Sustituye las apariciones de patron en cadena con repl
<code>escape(cadena)</code>	Prepara una cadena convirtiéndola en otra con todos los caracteres de escape donde sean necesarios

Funciones del módulo `re`

- Función `compile`(patron[, flags])
 - Transforma una ER escrita como una cadena en un objeto de tipo expresión regular.
 - Los objetos de tipo expresión regular tienen a las funciones `search` y `match` como métodos.
- Función o método `search`(patron, cadena[, flags])
 - Busca en la cadena especificada la primera aparición del patrón si es que se encuentra. Devuelve un `MatchObject` (que sirve como `True`) o `None` si no lo encuentra
 - `re.search('?ola', "que tal, hola")`

Funciones del módulo `re`

- Función `match`(patron, cadena[, flags])
 - Identifica el patrón al principio de la cadena, devuelve un objeto `MatchObject` (que sirve como `True`) o `None` si no lo encuentra
- Función `fullmatch`(patron, cadena[, flags])
 - Identifica el patrón en toda la cadena, devuelve un objeto `MatchObject` (que sirve como `True`) o `None` si no lo encuentra
- Función `split`(patron, cadena[, maxsplit=0])
 - Trocea una cadena por el patrón (equivalente al `split` sobre cadenas), devuelve una lista

Funciones del módulo **re**

```
import re

if re.search('hola','hakshfsahfholadjkshf'):
    print ('Acierto')
else:
    print('Error')

if re.match('hola','hakshfsahfholadjkshf'):
    print ('Acierto')
else:
    print('Error')

if re.search('hola','holaakshfsahfholadjkshf'):
    print ('Acierto')
else:
    print('Error')

if re.match('hola','holaakshfsahfholadjkshf'):
    print ('Acierto')
else:
    print('Error')
```

Funciones del módulo **re**

- Función **findall**(patron, cadena[, flags])
 - Devuelve una lista de cadenas con todas las apariciones no solapadas del patrón en la cadena
- Función **sub**(patron, repl, cadena[, count=0])
 - Devuelve la cadena obtenida reemplazando las apariciones (no solapadas y por la izquierda) del patrón en la cadena con repl.

Funciones del módulo **re**

- Función **escape**(cadena)
 - Devuelve una cadena todos los caracteres excepto letras, números, preparados para formar parte de ER es decir con \ delante

```
>>>import re
>>>x=re.escape('hola[^adios')
>>>print(x)
hola\[^\^adios
```


Módulo re: Flags

Método	Descripción
re.I re.IGNORECASE	Identifica sin considerar la diferencia mayúsculas minúsculas
re.M re.MULTILINE	El carácter ^ identifica no solo al principio de cadena sino también al principio de línea. Similarmente \$ identifica al final de cadena y de línea.
re.S re.DOTALL	. Detecta todo incluso nueva línea
re.X re.VERBOSE	Permite escribir ER más claras. Los espacios en blanco se ignoran, excepto en clases de caracteres o precedidos por \ único. Cuando una línea contiene # fuera de una clase de caracteres o con \, todos los caracteres desde el # al final de línea se considera comentario

Funciones del módulo **re**

Ejercicio 14:

Crear una función que reciba una cadena de ADN y la convierta en una de ARN.

Conjuntos de caracteres

- Permiten a un conjunto de caracteres, en vez de a uno solo, aparecer en un punto dado de una ER.
- Los conjuntos de caracteres se representan mediante corchetes [], donde lo que aparece dentro de los corchetes son los caracteres que se quiere hacer coincidir.
- Ej:
 - [bcr]at identificaría “bat”, “cat” o “rat”
 - [yY][eE][sS] identificaría yes, sin preocuparse de mayúsculas y minúsculas.
- Rangos. Dentro de los conjuntos de caracteres se puede usar rangos, especificados con –
- [a-z] identifica cualquier carácter alfabético en minúsculas
- [a-zA-Z0-9] identifica cualquier carácter alfabético o numérico

Grupos

- `()` identifica la ER que esté dentro de los paréntesis.
- Indica el comienzo y fin de un grupo identificado
- Los contenidos del grupo pueden conocerse con **match**
- Los contenidos del grupo pueden usarse después en la cadena con el carácter especial `\numero`
- Los grupos se numeran por su paréntesis izquierdo
- El grupo 0 es todo el patrón

Alternancia

- Se expresan mediante el pipe: |
 'python|perl' identifica o python o perl
 'p(ython|erl)' identifica lo mismo

Repeticiones

- Los caracteres de repetición se colocan después del carácter, conjunto de caracteres o grupo que se quiere repetir
 - * repetición del patrón anterior cero o más veces , sirve para indicar que algo es opcional, puede aparecer o no.
 - + repetición del patrón una o más veces.
 - ? repetición del patrón cero o una vez.
 - {n} repetición del patrón n veces.
 - {n,} repetición del patrón n o más veces.
 - {,m} repetición del patrón entre 0 y m veces
 - {n,m} repetición del patrón entre n y m veces.

Repeticiones

- Los caracteres de repetición tratan de identificar lo más posible de la cadena. Si se desea que un carácter de repetición identifique lo menos posible, debe escribirse con una ? al final.
 - a+? Trata de identificar la letra a 1 o más veces, comenzando por 1 vez.

Ejercicio 15:

Para el siguiente código:

```
re.match('^.*)(cat)(.*)', 'the cat in the hat')
```

Determinar qué se identifica en cada uno de los grupos.

Hacer lo mismo con:

```
re.match('^.*)(at)(.*)', 'the cat in the hat')
```

Anclajes

- ^ busca una subcadena al principio de una cadena
'^http'
- \$ busca una subcadena al final de una cadena
'org\$'
- '^http\$' identifica solo http

Ejercicios

Ejercicio 16:

En PERL los nombres de variable se forman igual que en Python pero van precedidos del símbolo \$.

Crear un programa que pida al usuario una palabra y diga si es una variable legal de PERL

Ejercicio 17:

En Python los números en punto flotante pueden aparecer con los siguientes formatos:

1.23

1.

3.14e-10

4E21

4.0e+45

Crear un programa que pida al usuario un número y diga si es un número en punto flotante

Objetos MatchObject y grupos

- Las funciones del módulo **re** que tratan de identificar un patrón en una cadena devuelven objetos MatchObject cuando se identifica algo.
- Estos objetos contienen información sobre la subcadena que se ha identificado y sobre qué grupos se identificaron

Objetos MatchObject y grupos

- Métodos sobre ER:

Método	Descripción
<code>regex.search(cadena[, pos[, endpos]])</code>	Busca la ER regex en la cadena, y devuelve un objeto MatchObject. Pos indica donde empezar la búsqueda y endpos donde acabarla
<code>regex.match(cadena[, pos[, endpos]])</code>	Busca la ER regex al principio de la cadena, y devuelve un objeto MatchObject. Pos indica donde empezar la búsqueda y endpos donde acabarla
<code>regex.findall(cadena[, pos[, endpos]])</code>	Igual a la función findall pero usando la ER compilada
<code>regex.split(cadena[, maxsplit=0])</code>	Igual a la función split pero usando a ER compilada
<code>regex.sub(repl, cadena, count=0)</code>	Igual a la función sub pero usando a ER compilada

Objetos MatchObject y grupos

```
prog = re.compile('h[0-9]\.org')  
result = prog.match('esto es un dato h4.org valido'))
```

es equivalente a:

```
result = re.match('h[0-9]\.org', 'esto es un dato h4.org valido'))
```

Objetos MatchObject y grupos

- Métodos sobre MatchObjects:
- match es un objeto MatchObject

Método	Descripción
<code>match.group([group1, ...])</code>	Muestra lo identificado en los grupos indicados. Si no se indica grupo devuelve el grupo 0, es decir todo lo identificado
<code>match.groups()</code>	Devuelve una tupla que contiene todos los subgrupos identificados
<code>match.start([group])</code>	Devuelve la posición de comienzo de lo identificado en el grupo
<code>match.end([group])</code>	Devuelve la posición de finalización (+1) de lo identificado en el grupo
<code>match.span([group])</code>	Devuelve el comienzo y fin del grupo

Objetos MatchObject y grupos

```
>>>m=re.match('www\.(.*)\.{3}', 'www.python.org')
>>>m.group(1)
'python'
>>>m.start(1)
4
>>>m.end(1)
10
>>>m.span(1)
(4,10)
```

Ejercicios

Ejercicio 18:

Crear un programa que abra el fichero GenBank.gb y haga lo siguiente:

1. Guarde en un fichero toda la información que no se corresponda con los datos de una secuencia de ADN.
2. Guarde en otro fichero los datos de secuencia.
3. Analice cada línea de secuencia e indique en qué líneas aparece tca y en qué posición dentro de la línea.

Módulo os.path

Módulo os.path

- Módulo que contiene funciones sobre nombres de paths

`os.path.exists(path)` dice si ese path existe. Sirve para directorios y ficheros

`os.path.getatime(path)` devuelve el momento del último acceso a path

`os.path.getsize(path)` devuelve el tamaño en bytes de path

`os.path.isfile(path)` devuelve True si path es un fichero

`os.path.isdir(path)` devuelve True si path es un directorio

`os.path.split(path)` devuelve una lista (cabeza,cola) con el directorio en cabeza y el fichero en cola

Módulo os

- Módulo que contiene funciones del sistema operativo

os.**chdir**(path) cambia el current directory a path

os.**getcwd**() devuelve el current directory

os.**listdir**(path) devuelve una lista con todas las entradas del directorio path

Ejercicios

Ejercicio 19:

Crear un programa pida al usuario el nombre de un directorio, busque en este directorio todos los ficheros de tipo GenBank (extension.gb), los abra y diga de qué organismo contiene datos.

Mejorarlo para que mire también en los subdirectorios del directorio dado.

Ejercicio 20:

Queremos almacenar los artículos que leemos en una base de datos (BD), pero no nos gusta ninguna de las disponibles, por lo tanto vamos a construirla nosotros mismos.

La BD será una lista de diccionarios. Cada diccionario consta de 5 campos: Título del artículo, autores, revista, fecha y el nombre del fichero donde guardamos un resumen del artículo.

El programa nos permitirá hacer una serie de cosas, elegidas por un menú:

- 1.- Introducir un nuevo elemento, esto implicará salvarlo en un archivo.
- 2.- Listar todos los artículos, especificando los 4 primeros campos.
- 3.- Buscar si existe un artículo dando una palabra clave del título.
- 4.- Buscar si existe un artículo dando el nombre de un autor
- 5.- Listar todos los artículos de una determinada revista

Cada vez que se arranca el programa deberán recuperarse los datos almacenados previamente.