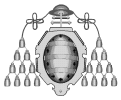

Universidad de Oviedo
Grado en Física

Introducción a la Física Computacional

Rosario Díaz Crespo, Julio Manuel Fernández Díaz

Curso 2015–16

Python como lenguaje algorítmico: Primera parte

Práctica 4

Python como lenguaje algorítmico: Primera parte

4.1. Objetivo

Familiarizarse con el intérprete de Python y con el uso de las variables, los tipos datos y los operadores en Python.

4.2. Introducción

4.2.1. ¿Qué es Python?

Python es un lenguaje de programación creado a principios de los años 90 por Guido van Rossum. Se trata de un lenguaje de programación de alto nivel, interpretado o de *script*, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos.

Lenguaje interpretado o de script

Un lenguaje interpretado o de *script* es aquél que se ejecuta utilizando un programa intermedio llamado intérprete, a diferencia de un lenguaje compilado que traduciría el código a un lenguaje máquina para que pudiera ser ejecutado directamente por un ordenador. La ventaja de los lenguajes compilados es que su ejecución es más rápida, sin embargo los lenguajes interpretados son más flexibles y más portables.

Python tiene, no obstante, muchas de las características de los lenguajes compilados, por lo que se podría decir que es semi-interpretado. En Python el código fuente se traduce a un pseudocódigo máquina intermedio (llamado *bytecode*) la primera vez que se ejecuta, generando archivos `.pyc` o `.pyo`, que son los que se ejecutarán en sucesivas ocasiones.

Tipado dinámico

La característica de tipado dinámico se refiere a que no es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se le asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo. Por ejemplo: `x = 3.0`, asigna a `x` un número real, si a continuación se escribe `x = log`, se asigna a `x` la función `log`; a partir de ese momento `x(5.0)` nos calculará el logaritmo neperiano de `5.0`.

Fuertemente tipado

Al ser fuertemente tipado no se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si tenemos una variable que contiene un texto (tipo cadena) no podremos tratarla como un número (sumar la cadena de caracteres `'2'` con el el número `4` está prohibido).

Multiplataforma

El intérprete de Python está disponible en multitud de plataformas (Unix, Solaris, Linux, MS/Dos, Windows, Mac OS, etc.) por lo que si no utilizamos bibliotecas específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas con pocos cambios.

Python es software libre

No hay una empresa comercial detrás, sino toda la comunidad internacional. Por ello, no está sujeto a los caprichos de una compañía. Eso no quiere decir que no podamos realizar desarrollos comerciales en Python. Excepto que usemos una biblioteca de módulos que lo cite expresamente, una aplicación que desarrollemos en Python puede ser vendida y reportar beneficios.

Orientado a objetos y funcional

La orientación a objetos (OO) es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones entre los objetos.

La programación funcional (FP de sus siglas en inglés) es otro paradigma avanzado de programación.

En el presente curso no desarrollaremos los aspectos de OO ni la FP.

Módulos prefabricados

Python tiene una ventaja respecto a otros lenguajes debido a que posee infinidad de módulos prediseñados (quizá sea el que más tiene). Cada módulo contiene herramientas de programación para realizar tareas relacionadas entre sí. Se puede usar un módulo (conociendo su funcionamiento) de una manera simple, y el diseño de nuevos módulos es sencillo.

Algunos módulos prefabricados son:

- Matemáticas (`math`, `numpy`, `scipy`, `sympy`, etc.).
- Estadística (`stats`, etc.).
- Representaciones gráficas 2D y 3D (`matplotlib`, `visual`, `Gnuplot`, etc.).
- Interfaces gráficas (`tkinter`, `wxPython`, etc.).
- Bases de datos (`pySQLite`, etc.).
- Tratamiento de imágenes (`PIL`, etc.).
- Manipulación de bases de datos científicas como (`NetCDF`, etc.).
- Servidor de aplicaciones web (`zope`) —no es exactamente un módulo.

4.2.2. ¿Por qué Python?

Python es uno de los mejores lenguajes para comenzar a programar debido a que su curva de aprendizaje es poco empinada. Su aprendizaje puede ser realizado incrementalmente de manera ventajosa (dejando, por ejemplo, la programación FP y OO para el final). Como ya se ha mencionado anteriormente está disponible en muchas plataformas y contiene bibliotecas de programas para casi cualquier tema que se nos ocurra. Además, puede combinarse fácilmente con lenguajes compilados (C, C++, Fortran) para las tareas que requieran velocidad de cálculo.

Ventajas de Python

- Su sintaxis es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo.
- El tipado es dinámico, lo que nos evita tener que estar continuamente definiendo variables y asignando memoria.
- El gestor de memoria simplifica el uso de variables (la memoria se libera automáticamente cuando no se necesita).
- La gran cantidad de bibliotecas disponibles y la potencia del lenguaje, hacen que desarrollar una aplicación en Python sea sencillo y muy rápido.

Comparación de Python con otros lenguajes

- Python es un lenguaje completo y con sintaxis homogénea y coherente (a diferencia de Matlab® y Octave, v.g.).
- Es más lento que los lenguajes compilados pero es mucho más rápido desarrollar las aplicaciones.
- Es más fácil realizar el mantenimiento de programas que con otros lenguajes (con el paso del tiempo todos los programas tienen errores que eliminar).

4.3. Usando Python en modo interactivo

En modo interactivo Python se puede usar como una calculadora. Para invocar el modo interactivo de Python teclee en el terminal:

```
$ python
```

```
>>>
```

El intérprete nos indica con `>>>` (se denomina prompt en inglés) que está esperando una orden del usuario.

Ejemplo 4.3.1. *Utilice Phyton como una calculadora:*

1. *Invoque el modo interactivo de Python.*
2. *Escriba 3+4 y pulse <Intro>. En la pantalla aparecerá:*

   ```
   >>> 3+4
   7
   ```

3. *A continuación asigne a r el valor de 3 y calcule s como el área de un círculo de radio r, de la forma:*

   ```
   >>> r = 3; s = 3.1416*r**2
   ```

   (Nota: `**` indica exponenciación.)

4. *Teclee s y pulse <Intro> para que muestre el resultado de la operación de la forma:*

   ```
   >>> s
   28.2744
   ```

Ejercicio 4.3.1. Se lanza una pelota hacia arriba con una velocidad inicial de 20 m/s.

1. ¿Cuánto tiempo está la pelota en el aire? (Desprecie la altura del punto de lanzamiento)
2. ¿Cuál es la mayor altura alcanzada por la pelota?

4.4. Python en modo no interactivo

El modo no interactivo se usa indicándole al intérprete qué fichero queremos usar para ejecutar en Python.

Ejemplo 4.4.1. *Trabaje con Python en modo no interactivo.*

1. *Cree un fichero que se llame hola.py y que contenga:*

   ```
   #----- hola.py ----
   print('Hola amigo')
   ```

2. *Teclee en la teminal (pulsando <Intro> al final de la línea):*

   ```
   $ python hola.py
   ```

   La respuesta será:

   ```
   Hola amigo
   $
   ```

En el modo no interactivo Python no se puede usar como una calculadora (no imprime las operaciones directamente), y se debe usar `print` para visualizar resultados.

Ejercicio 4.4.1. Cree un fichero con el nombre `pelota.py` que resuelva el problema del ejercicio anterior.

4.5. Variables y tipos de datos

En la mayoría de los lenguajes el nombre de una variable representa un tipo y un valor almacenado en una posición fija de memoria. El valor puede cambiarse pero el tipo no.

Esto no ocurre en Python donde ya hemos dicho que las variables tienen tipos dinámicos.

Los nombres de las variables, denominados **identificadores**, en Python pueden contener letras, números y el carácter _ (barra baja). No pueden contener letras acentuadas ni la ñ. Se distinguen mayúsculas y minúsculas.

Ejemplos:

`aviso`, `x`, `f7`, `Fecha_de_nacimiento`

No pueden empezar por un número y es recomendable no empezarlos por _. No valdrían:

`5solos`, `año`, `carbón`, `acel-gravedad`

Existen algunas palabras que no se pueden usar como nombres de variables. Se llaman **palabras reservadas**, las cuales tienen utilidad específica en el lenguaje. Ejemplos:

`if`, `while`, `for`, `try`

Los **tipos de datos** con los que podemos trabajar en Python son:

- Valores booleanos (lógicos).
- Números: enteros, reales, complejos.
- Cadenas de caracteres (*strings*).
- Colecciones de datos.

Asociar un valor determinado a una variable se denomina **asignación**, la cual utiliza el carácter `=`. Por ejemplo, podemos asociar a la variable `n` el valor entero `10` mediante:

```
>>> n = 10
```

A partir de ese momento, hasta el final del programa[1] o hasta que hagamos otra asignación a `n`, una referencia a `n` usa el valor entero `10`.

4.5.1. Valores booleanos

Una variable de tipo booleano o lógico sólo puede tener dos valores:

- `True` (verdadero).
- `False` (falso).

Estos valores son especialmente importantes para las expresiones condicionales y los bucles, como veremos más adelante.

Ejemplo 4.5.1. *Escriba:*

```
>>> k = True
```

```
>>> type(k)
```

4.5.2. Enteros

Los números enteros son aquellos números positivos o negativos que no tienen decimales (incluido el cero). En Python se pueden representar mediante:

- `int` (de integer, entero)
- `long` (largo).

[1] No exactamente. Ya hablaremos más adelante de la visibilidad de las variables en fragmentos de código dentro de un programa.

Enteros: tipo int

En una variable de tipo `int` de Python podemos almacenar con 32 bits números de -2^{31} a $2^{31}-1$, o lo que es lo mismo, de -2147483648 a 2147483647 (alrededor de dos mil millones).

En plataformas de 64 bits, el rango es desde -2^{63}, es decir, -9223372036854775808 hasta $2^{63}-1$, que equivale a 9223372036854775807 (alrededor de nueve trillones).

Enteros: tipo long

El tipo `long` de Python permite almacenar números de cualquier precisión, estando limitados solo por la memoria disponible en la máquina. Al asignar un número entero a una variable esta pasará a tener tipo `int`, a menos que el número sea tan grande como para requerir el uso del tipo `long`.

Ejemplo 4.5.2. *Tipos de enteros.*

1. *Invoque el modo interactivo de Python.*
2. *Teclee:*
   ```
   >>> i = 3
   ```
3. *Teclee:*
   ```
   >>> i
   ```
   la respuesta será:
   ```
   3
   ```
4. *Escriba:*
   ```
   >>> type(i)
   ```
   la respuesta será:
   ```
   <type 'int'>
   ```
   La funcíon type nos muestra el tipo de dato.

1. *Teclee:*
   ```
   >>> i = 2**62
   ```
2. *Teclee:*
   ```
   >>> i
   ```
   la respuesta será:
   ```
   4611686018427387904
   ```

3. *Escriba:*

   ```
   >>> type(i)
   ```

   la respuesta será:

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

4. *Teclee:*

   ```
   >>> i = 2**63
   ```

5. *Teclee:*

   ```
   >>> i
   ```

   la respuesta será:

   ```
   9223372036854775808L
   ```

6. *Escriba:*

   ```
   >>> type(i)
   ```

   la respuesta será:

   ```
   <type 'long'>
   ```

4.5.3. Reales: tipo float

Los números reales son los que tienen decimales. En Python se expresan mediante el tipo `float`. Python implementa su tipo `float` utilizando 64 bits y sigue el estándar IEEE 754: 1 bit para el signo, 11 para el exponente, y 52 para la mantisa. Esto significa que los valores que podemos representar van

desde $\pm 4.94065645841246544 \times 10^{-324}$ hasta $\pm 1.7976931348623157 \times 10^{308}$

aunque las operaciones con números con valor absoluto por debajo de 2.225×10^{-308} (números 'subnormalizados') tienen errores de redondeo y no se recomienda su uso.

Ejemplo 4.5.3. *Trabajar con números reales.*

1. *Teclee:*

   ```
   >>> r = 1.0
   >>> r
   ```

   la respuesta será:

   ```
   1.0
   ```

2. *Escriba:*

   ```
   >>> type(r)
   ```

   la respuesta será:

   ```
   <type 'float'>
   ```

   Se puede también utilizar el formato científico.

3. *Teclee:*

   ```
   >>> r = 1.234e5
   >>> r
   ```

   la respuesta será:

   ```
   123400.0
   ```

   Como vemos, el número real 1.234×10^5 *puede teclearse* `1.234e5` *(y también* `123400.0`*).*

4. *Escriba:*

   ```
   >>> type(r)
   ```

   la respuesta será:

   ```
   <type 'float'>
   ```

5. *Teclee:*

   ```
   >>> x = 1e308
   >>> x
   ```

   la respuesta será:

   ```
   1e+308
   ```

6. *Teclee:*

   ```
   >>> x = 2e308
   >>> x
   ```

   la respuesta será:

   ```
   inf
   ```

7. *Escriba:*

   ```
   >>> x = -2e308
   >>> x
   ```

   la respuesta será:

   ```
   -inf
   ```

8. *Escriba:*

   ```
   >>> x = 5e-324
   >>> x
   ```

   la respuesta será:

   ```
   5e-324
   ```

9. *Escriba:*

   ```
   >>> x = 5e-325
   >>> x
   ```

   la respuesta será:

   ```
   0.0
   ```

10. *Escriba:*

```
>>> x = -5e-325
>>> x
```

la respuesta será:

```
-0.0
```

11. *Realice asignaciones con los siguientes valores: 3e-324, 6e-324, 6e-325, 9e-325 y compruebe los errores de redondeo.*

Ejercicio 4.5.1. Si se quiere resolver el ejercicio 4.3.1 de forma mas genérica se deben definir v_0, g, t e y como variables en el programa, inicializar los datos y obtener las incógnitas. Cree un fichero con el nombre `pelota_var.py` que resuelva nuevamente el problema utilizando variables.

4.5.4. Complejos

Los números complejos son aquellos que tienen parte imaginaria. Este tipo se denomina `complex` en Python, y se almacena usando dos valores `float`, debido a que estos números son una extensión de los números reales. Se utiliza `j` para indicar la unidad imaginaria.

Ejemplo 4.5.4. *Trabajar con números complejos.*

1. *Escriba:*

```
>>> c = 1.0+2j
>>> c
```

la respuesta será:

```
(1+2j)
```

2. *Escriba:*

```
>>> type(c)
<type 'complex'>
```

4.5.5. Cadenas de caracteres (strings)

Una cadena es una secuencia de caracteres entre comillas simples `'` o dobles `"`. Dentro de las comillas se pueden añadir caracteres especiales como:

- $\backslash$ (carácter de continuación)
- $\backslash n$ (carácter de nueva línea)
- $\backslash t$ (carácter de tabulación)

Ejemplo 4.5.5. *Para definir una cadena:*

1. *Introduzca:*

   ```
   >>> c = "Hola mundo"
   ```

2. *Escriba:*

   ```
   >>> print(c)
   ```

   la respuesta será:

   ```
   Hola mundo
   ```

3. *Teclee:*

   ```
   >>> type(c)
   ```

   la respuesta será:

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

También es posible encerrar una cadena entre triples comillas (' o dobles "). De esta forma podremos escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que introdujimos sin tener que recurrir al carácter $\backslash n$.

Ejemplo 4.5.6. *Para definir una cadena con varias líneas:*

Escriba:

```
>>> g = '''esto
... es
... un
... ejemplo'''
>>> g
'esto\nes\nun\nejemplo'
```

El operador + se utiliza para unir cadenas y el operador : se utiliza para extraer una porción de cadena.

Ejemplo 4.5.7. *Operar con cadenas.*

1. *Escriba:*

   ```
   >>> cadena1 = 'Presiona return para salir '
   >>> cadena2 = 'del programa'
   >>> print cadena1 +' '+ cadena2
   ```

   La respuesta será:

   ```
   Presiona return para salir del programa
   ```

2. *Podemos seleccionar un trozo de la cadena, para ello escriba:*

   ```
   >>> print(cadena1[0:15])
   ```

   La respuesta será:

   ```
   Presiona return
   ```

3. *Podemos asignar a una variable un trozo de cadena, para ello haga:*

   ```
   >>> cad = cadena1[-5:]
   >>> cad
   ```

   `[-5:]` *indica los últimos 5 caracteres. La respuesta será:*

   ```
   salir
   ```

4. *Existen funciones que operan sobre cadenas. Por ejemplo, convertir a mayúsculas. Si escribe:*

   ```
   >>> cad.upper()
   ```

   la respuesta será:

   ```
   SALIR
   ```

Una cadena es un objeto inmutable, no puede ser modificado con una declaración, y tiene una longitud fija. El intento de violar su inmutabilidad dará un error.

Ejemplo 4.5.8. *Para saber la longitud de las cadenas:*

1. *Escriba:*

   ```
   >>> s = 'Presiona return para salir'
   >>> len(s)
   ```

   la respuesta será:

   ```
   26
   ```

2. *No se puede cambiar un trozo de una cadena, si escribe:*

   ```
   >>> s[0] = 'p'
   ```

   la respuesta nos dará un error.

4.6. Operadores aritméticos

Operador	Descripción	Ejemplo	
+	Suma	r=2+3	r es 5
-	Resta	r=4 - 7	r es -3
-	Negación	r= -7	r es -7
*	Multiplicación	r=2*6	r es 12
**	Exponente	r=2**6	r es 64
/	División	r=3.5/2	r es 1.75
//	División entera	r=3.5//2	r es 1.0
%	Módulo	r=7 %2	r es 1

Si una operación aritmética incluye números de diferentes tipos, el resultado se convierte automáticamente a un tipo después de la operación según el siguiente esquema:

`bool` (`True`≡`1`, `False`≡`0`) → `int` → `long` → `float` → `complex`

Ejemplo 4.6.1. *Operación aritmética con diferentes tipos de datos.*

1. *Teclee:*

   ```
   >>> a = 2
   ```

   La asignación `a = 2` *crea una asociación entre el nombre* `a` *y el valor entero* `2` *y* `b = 4.0` *crea una asociación entre el nombre* `b` *y el valor real* `4.0`*, puede comprobarlo con la función* `type`*:*

   ```
   >>> type(a)
   >>> type(b)
   ```

2. *Escriba:*

   ```
   >>> print(a)
   2
   >>> print(b)
   4.0
   >>> c= a*b
   >>> print(c)
   8.0
   ```

Ejemplo 4.6.2. *Con una operación aritmética se puede cambiar el tipo de datos.*

1. *Teclee:*

   ```
   >>> b = 2
   ```

   La asignación `b = 2` *crea una asociación entre el nombre* `b` *y el valor entero* `2`*. Por lo tanto ahora* `b` *es un número entero, puede comprobarlo con la función* `type`*:*

   ```
   >>> type(b)
   ```

2. *Escriba:*

   ```
   >>> print(b)
   2
   >>> b = b * 2.0
   ```

   `b*2.0` *asocia el resultado con* `b` *eliminando la asociación original con el entero* `2`*. Después de esta declaración* `b` *se refiere a un tipo coma flotante de valor* `4.0`*.*

   ```
   >>> print(b)
   4.0
   ```

   puede comprobarlo con la función `type`*.*

3. *Teclee:*

   ```
   >>> type(b)
   ```

Ejercicio 4.6.1. Cree un fichero con el nombre `temperatura.py` que calcule la temperatura en la escala Fahrenheit equivalente a 21°C, de acuerdo con la expresión:

$$F = \frac{9}{5}C + 32,$$

donde C representa la temperatura Celsius. Compruebe el resultado obtenido haciendo la misma operación con la calculadora.

Ejemplo 4.6.3. *Algunas de estas operaciones también están definidas para cadenas (pero significan otra cosa, evidentemente).*

1. *Escriba:*

   ```
   >>> s = 'Hello '
   >>> t = 'to you'
   >>> print(3*s)
   ```

   la respuesta será:

   ```
   Hello Hello Hello
   ```

2. *Escriba:*

   ```
   >>> print(s + t)
   ```

   la respuesta será:

   ```
   Hello to you
   ```

   pero cuidado, no todas las operaciones están permitidas:

3. *Si escribe:*

   ```
   >>> print(3+s)
   ```

   obtendrá una respuesta de error.

Las versiones posteriores a Python 2.0 han aumentado la asignación de los operadores, de manera que resulten familiares a los usuarios de C. A la izquierda el operador, a la derecha la equivalencia

a += b	a = a+b
a -= b	a = a-b
a *= b	a = a*b
a /= b	a = a/b
a **= b	a = a**b
a %= b	a = a %b

4.7. Operadores relacionales

Los operadores relacionales devuelven `True` si es cierto y `False` si es falso.

Operador	Descripción	Ejemplo	
$<$	Menor que	$5 < 3$	False
$>$	Mayor que	$5 > 3$	True
$<=$	Menor o igual	$5 <= 5$	True
$>=$	Mayor o igual	$5 >= 3$	True
$==$	Igual a	$5 == 3$	False
!=	Distinto a	5 != 3	True

Ejemplo 4.7.1. *Trabajar con operadores relacionales.*

1. *Teclee:*

```
>>> a = 2
>>> b = 1.99
>>> c = '2'
>>> print(a > b)
```

la respuesta es:

```
True
```

2. *Escriba:*

```
>>> print(a == c)
```

la respuesta es:

```
False
```

4.8. Operadores condicionales

Los operadores condicionales devuelven `True` si es cierto y `False` si es falso.

Operador	Descripción	Ejemplo	
and	¿se cumple a y b?	$5 > 3$ and $5 < 3$	False
or	¿se cumple a o b?	$5 > 3$ or $5 < 3$	True
not	no	not $5 <= 3$	True

Ejemplo 4.8.1. *Trabajar con operadores condicionales (continuación del ejemplo anterior).*

1. *Escriba:*

   ```
   >>> print((a > b) and (a != c))
   ```

   la respuesta es:

   ```
   True
   ```

2. *Escriba:*

   ```
   >>> print((a > b) or (a == b))
   ```

   la respuesta es:

   ```
   True
   ```

4.9. Colecciones

Además de los tipos básicos de datos que hemos visto, números, cadenas y booleanos, existen otros tipos de colecciones de datos:

- Tuplas.
- Listas.
- Diccionarios.

4.9.1. Tuplas

Una tupla es una secuencia de objetos arbitrarios separados por comas y encerrados entre paréntesis. Si la tupla contiene sólo un objeto, se requiere una coma al final, por ejemplo, x=(2,).

Los elementos de las tuplas se numeran empezando con `0` siempre. Si la tupla tiene `n` elementos se numeran `0`, `1`, `2`, ..., `n-1`.

Además se pueden usar índices negativos: `-1` es equivalente a `n-1`, `-2` lo es a `n-2`, etc.

Las tuplas soportan las mismas operaciones que las cadenas, son también inmutables.

Ejemplo 4.9.1. *Crear una tupla.*

1. *Escriba:*

   ```
   >>> t = (1)
   >>> type(t)
   ```

   el resultado es:

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

2. *Teclee:*

   ```
   >>> t = (1,)
   >>> type(t)
   ```

   el resultado es:

   ```
   <type 'tuple'>
   ```

Las tuplas, al igual que las cadenas, forman parte de un tipo de objetos llamados secuencias, por ello podemos utilizar el operador `[]` para referirse a sus elementos.

Ejemplo 4.9.2. *Referirse a elementos de una tupla.*

1. *Escriba:*

   ```
   >>> t = (1, 2, 3)
   >>> type(t)
   ```

   la respuesta es:

   ```
   <type 'tuple'>
   ```

2. *Teclee:*

   ```
   >>> mi_var = t[0]
   ```

   Observe la respuesta.

3. *Teclee:*

   ```
   >>> m_var = t[0:2]
   ```

   Observe la respuesta.

Ejemplo 4.9.3. *Operaciones con tuplas.*

1. *Cree la siguiente tupla:*

   ```
   >>> rec = ('Smith', 'John', (6, 23, 68))
   ```

2. *Podemos asignar variables a los elementos de una tupla, para ello teclee:*

   ```
   >>> apellido, nombre, fecha_nac = rec
   >>> print(nombre)
   ```

   la respuesta es:

   ```
   John
   ```

3. *Escriba:*

   ```
   >>> a_nac = fecha_nac[2]
   ```

   el operador `[]` *nos permite referirnos a un elemento de la tupla fecha_nac y asignárselo a la nueva variable a_nac.*

4. *Teclee:*

   ```
   >>> print(a_nac)
   ```

   la respuesta es:

   ```
   68
   ```

5. *Podemos concatenar los elementos de una tupla, para ello escriba:*

   ```
   >>> nombre = rec[1] + ' ' + rec[0]
   >>> print(nombre)
   ```

   la respuesta es:

   ```
   John Smith
   ```

6. *Para seleccionar algunos elementos de la tupla, teclee:*

   ```
   >>> print(rec[0:2])
   ```

   la respuesta es:

   ```
   ('Smith', 'John')
   ```

4.9.2. Listas

Una lista es similar a una tupla, pero puede ser modificada, de manera que pueden cambiarse sus elementos y su longitud.

Una lista se identifica encerrándola entre corchetes. Sería equivalente a lo que en otros lenguajes se conoce por *arrays* y vectores.

Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos, etc., y también listas. Los elementos de las listas se numeran empezando con 0 siempre.

Ejemplo 4.9.4. *Crear una lista.*

1. *Cree una lista de números reales, para ello escriba:*

   ```
   >>> a = [1.0, 2.0, 3.0]
   >>> type(a)
   ```

   la respuesta es:

   ```
   <type 'list'>
   ```

Se pueden realizar operaciones con listas, algunas de ellas son:

- `len(s)` nos devuelve la longitud de la lista s.
- `s.append(x)` añade x al final de la lista s.
- `s.insert(k,x)` inserta x en la posición k de la lista s.
- `del s[k]` elimina de la lista s el elemento en la posición k.
- `s[i:j]` nos devuelve una sublista desde el elemento `i` hasta el `j-1`.
- `[x]*n` genera una lista de longitud n cuyos elementos son x.

Ejemplo 4.9.5. *Operaciones con listas.*

1. *Si quiere añadir el elemento 4.0 a la lista escriba:*
   ```
   >>> a.append(4.0)
   ```
2. *Lo comprobamos con:*
   ```
   >>> print(a)
   ```
   la respuesta es:
   ```
   [1.0, 2.0, 3.0, 4.0]
   ```
3. *Para insertar el caracter 'a' en la posición 0 de la lista se escribe:*
   ```
   >>> a.insert(0, 'a')
   >>> print(a)
   ```
   la respuesta es:
   ```
   ['a', 1.0, 2.0, 3.0, 4.0]
   ```
4. *Para saber la longitud de la lista escriba:*
   ```
   >>> print(len(a))
   ```
   la respuesta es:
   ```
   5
   ```
5. *Para modificar elementos de la lista:*
   ```
   >>> a[2:4] = [1.0, 1.0]
   ```
   con esta acción se han modificado los elementos 2 y 3 de la lista. Para comprobarlo:
   ```
   >>> print(a)
   ```
   la respuesta es:
   ```
   ['a', 1.0, 1.0, 1.0, 1.0, 4.0]
   ```

Ejemplo 4.9.6. *Operadores aritméticos con listas.*

1. *Cree una lista:*
   ```
   >>> a = [1, 2, 3]
   ```

2. *Escriba:*

   ```
   >>> print(3*a)
   ```

   la respuesta es:

   ```
   [1, 2, 3, 1, 2, 3, 1, 2, 3]
   ```

3. *Para concatenar listas:*

   ```
   >>> print(a + [4, 5])
   ```

   la respuesta es:

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

Si `a` es un objeto mutable, la declaración `b = a` no da como resultado un nuevo objeto `b`, simplemente crea una nueva referencia a `a`. cualquier cambio realizado a `b` se reflejará en `a`.

Para crear una copia independiente de una lista `a`, usaremos la declaración `c = a[:]`.

Ejemplo 4.9.7. *Copiar listas.*

1. *Cree una lista:*

   ```
   >>> a = [1.0, 2.0, 3.0]
   ```

2. *Se puede crear una nueva referencia para* `a`*, para ello escriba:*

   ```
   >>> b = a
   ```

3. *al cambiar un elemento de* `b`*:*

   ```
   >>> b[0] = 5.0
   ```

   el cambio se refleja en `a`*, lo comprobamos escribiendo:*

   ```
   >>> print(a)
   ```

   la respuesta es:

   ```
   [5.0, 2.0, 3.0]
   ```

4. *Para hacer una una copia de* `a`*:*

   ```
   >>> c = a[:]
   ```

   `c` *es copia independiente de* `a`*.*

5. *Para cambiar un elemento de* `c`*:*

   ```
   >>> c[0] = 1.0
   ```

6. *Se ha cambiado* `c`*, pero al ser una copia independiente de* `a` *ésta no ha cambiado; para comprobarlo:*

   ```
   >>> print(a)
   ```

   la respuesta es:

   ```
   [5.0, 2.0, 3.0]
   ```

En ocasiones es necesario emparejar dos listas, `[x1, x2, ...]` e `[y1, y2, ...]`, para formar otra lista con tuplas de la manera: `[(x1, y1), (x2, y2), ...]`. Obviamente la longitud de las dos listas iniciales debe ser la misma. Esto se hace con la función `zip`, que también funciona con tuplas.

Ejemplo 4.9.8. *Uso de la función* `zip`

```
l = [1, 2, 3, 4]
m = ['a', 'b', 'c', 'd']
lm = zip(l, m) # da: [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

Ejemplo 4.9.9. *Para deshacer el empaquetado en parejas anterior se usa también la función* `zip` *pero con un asterisco,* `*`*, de la manera siguiente:*

```
z = zip(*lm) # da: [(1, 2, 3, 4), ('a', 'b', 'c', 'd')]
```

Como vemos devuelve una lista con dos tuplas, separando de nuevo las listas originales.

Podemos representar matrices mediante listas anidadas, donde cada fila se representa por una lista. Salvo algunas excepciones no utilizaremos listas para crear matrices numéricas. Para crear matrices es mas conveniente utilizar el módulo `numpy` (que veremos más adelante).

Ejemplo 4.9.10. *Crear matrices utilizando listas.*

1. *Para crear una matriz escriba:*

   ```
   >>> a = [[1, 2, 3], \
   ... [4, 5, 6], \
   ... [7, 8, 9]]
   ```

   Recuerde que el carácter `\` *representa continuación en Python.*

2. *Compruebe la matriz creada:*

   ```
   >>> print(a)
   ```

   la respuesta es:

   ```
   [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   ```

   Las secuencias en Python tienen un offset cero, de manera que `a[0]` *representa la primera fila,* `a[1]` *la segunda y así sucesivamente.*

3. *Para imprimir la segunda fila de la matriz escriba:*

   ```
   >>> print(a[1])
   ```

   la respuesta es:

   ```
   [4, 5, 6]
   ```

4. *Para imprimir el tercer elemento de la segunda fila escriba:*

   ```
   >>> print(a[1][2])
   ```

   la respuesta es:

   ```
   6
   ```

Ejercicio 4.9.1. Cree un archivo con el nombre `matriz.py` en el que construya una matriz 3*3 cuyos elementos sean los números del 1 al 9 y realize las siguientes operaciones:

1. Imprima por orden los números pares de la matriz.
2. Haga una copia de la matriz y sustituya en ella los números impares por 0.

Ejemplo 4.9.11. *Operaciones con listas y cadenas.*

1. *Se puede romper una cadena y crear una lista. Para ello se usa la orden* `split(delimitador)`. *Si no se especifica el delimitador se utiliza el espacio por defecto. Escriba:*

   ```
   >>> s = 'esto es una cadena'
   >>> s.split()
   ```

   la respuesta es:

   ```
   ['esto', 'es', 'una', 'cadena']
   ```

   escriba ahora:

   ```
   >>> s = 'esto\nes\nuna\ncadena'
   >>> s.split('\n')
   ```

   la respuesta es:

   ```
   ['esto', 'es', 'una', 'cadena']
   ```

2. *Se pueden concatenar los elementos de una lista para formar una cadena. Para ello cree una lista:*

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

3. *Ahora cree una cadena a partir de la lista uniendo sus elementos, para ello escriba:*

   ```
   >>> ''.join(l)
   ```

   la respuesta es:

   ```
   'abc'
   ```

4. *También se pueden unir los elementos de la lista utilizando otro elemento que los separe:*

   ```
   >>> '/=!'.join(l)
   ```

   une separando con /=! y la respuesta es:

   ```
   a/=!b/=!c
   ```

4.9.3. Diccionarios

Los diccionarios, también llamados "matrices asociativas", deben su nombre a que son colecciones que relacionan una clave y un valor.

Ejemplo 4.9.12. *Podemos crear un diccionario con algunas unidades derivadas del SI.*

1. *Cree un diccionario escribiendo:*

   ```
   >>> uniSI={'frecuencia': 'hercio', 'fuerza': 'newton',
               'presion': 'pascal', 'energia': 'julio', 'potencia': 'vatio'}
   ```

   Debemos fijarnos en las dos llaves, una al principio { *y otra al final* }.

2. *Compruebe que funciona, escribiendo:*

   ```
   >>> uniSI['frecuencia']
   ```

   la respuesta debe ser:

   ```
   'hercio'
   ```

3. *Escriba:*

   ```
   >>> uniSI['potencia']
   ```

   la respuesta debe ser:

   ```
   'vatio'
   ```

Ejercicio 4.9.2. Cree un diccionario con las unidades del SI para las magnitudes fundamentales y compruebe que dicho diccionario funcione.