TEMA II

DESARROLLO DE APLICACIONES DE PROPÓSITO GENERAL

"Un sueño no se hace realidad por arte de magia, necesita sudor, determinación y trabajo duro"
Colin Powell

2. FUNDAMENTOS DEL LENGUAJE

1. Introducción

Python es un lenguaje de programación potente y fácil de aprender. Tiene estructuras de datos de alto nivel muy eficientes y un simple pero efectivo sistema de programación orientado a objetos. Su tipado dinámico, junto a su naturaleza interpretada lo convierten en un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en muchas áreas, para la mayoría de las plataformas.

Permite dividir tu programa en módulos que pueden reutilizarse en otros programas de Python. Tiene una gran colección de módulos estándar que puedes utilizar como la base de tus programas. Algunos de estos módulos proporcionan cosas como entrada/salida de ficheros, llamadas a sistema, sockets e incluso interfaces a herramientas de interfaz gráfica como Tk.

Es un lenguaje interpretado, lo cual puede ahorrarte mucho tiempo durante el desarrollo ya que no es necesario compilar ni enlazar. El intérprete puede usarse interactivamente, lo que facilita experimentar con características del lenguaje, escribir programas desechables o probar funciones cuando se hace desarrollo de programas de abajo hacia arriba..

Python permite escribir programas compactos y legibles. Los programas en Python son típicamente más cortos que sus programas equivalentes en C, C++ o Java por varios motivos:

- Los tipos de datos de alto nivel permiten expresar operaciones complejas en una sola instrucción;

- La agrupación de instrucciones se hace mediante indentación en vez de llaves de apertura y cierre;
- No es necesario declarar variables ni argumentos.

Python también es apropiado como un lenguaje para extender aplicaciones modificables.

Empezaremos con los conceptos básicos y las funcionalidades del lenguaje de programación

2.Generalidades

1. Un programa python está formado por instrucciones que acaban en un caracter de "salto de línea".
2. El punto y coma ";" se puede usar para separar varias sentencias en una misma línea, pero no se aconseja su uso.
3. Una línea empieza en la primera posición, si tenemos instrucciones dentro de un bloque de una estructura de control de flujo habrá que hacer una indentación.
4. La indentación se puede hacer con espacios y tabulaciones pero ambos tipos no se pueden mezclar. Se recomienda usar 4 espacios.
5. La barra invertida "\" al final de línea se emplea para dividir una línea muy larga en dos o más líneas.
6. Las expresiones entre paréntesis "()", llaves "{}" y corchetes "[]" separadas por comas "," se pueden escribir ocupando varias líneas.
7. Cuando el bloque a sangrar sólo ocupa una línea ésta puede escribirse después de los dos puntos.
8. Se utiliza el caracter # para indicar los comentarios de una sola línea
9. Existen dos alternativas para introducir comentarios multilinea:
   - Comentar cada una de las líneas con el carácter #:
   - Utilizar triple comillas (simples o dobles) para generar una cadena multilínea: si bien este método es aceptado, no es ignorado por el intérprete

3. Ayuda en python

Una función fundamental cuando queremos obtener información sobre los distintos aspectos del lenguaje es help. Podemos usarla entrar en una sesión interactiva:

```
>>> help()
```

O pidiendo ayuda de una termino determinado, por ejemplo:

```
>>> help(print)
```

4. Literales

Los literales nos permiten representar valores. Estos valores pueden ser de diferentes tipos, de esta manera tenemos diferentes tipos de literales:

Literales numéricos

- Para representar números enteros utilizamos cifras enteras (Ejemplos: 3, 12, -23). Si queremos representarlos de forma binaria comenzaremos por la secuencia 0b (Ejemplos: 0b10101, 0b1100). La representación octal la hacemos comenzando por 0o (Ejemplos: 0o377, 0o7) y por último, la representación hexadecimal se comienza por 0x (Ejemplos: 0xdeadbeef, 0xfff).
- Para los números reales utilizamos un punto para separar la parte entera de la decimal (12.3, 45.6). Podemos indicar que la parte decimal es 0, por ejemplo 10., o la parte entera es 0, por ejemplo .001, Para la representación de números muy grandes o muy pequeños podemos usar la representación exponencial (Ejemplos: 3.14e-10,1e100).
- Por último, también podemos representar números complejos, con una parte real y otra imaginaria (Ejemplo: 1+2j)

Literales cadenas

Nos permiten representar cadenas de caracteres. Para delimitar las cadenas podemos usar el carácter ‘ o el carácter “. Con el carácter /, podemos escapar algunos caracteres, veamos algunos ejemplos:

\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)

\r ASCII Carriage Return (CR)
\t ASCII Horizontal Tab (TAB)
\v ASCII Vertical Tab (VT)

5. Variables y constantes

El nombre de una variable tiene que empezar por una letra o por el carácter _ seguido de letras, números o guiones bajos. No hay que declarar la variable antes de usarla, el tipo de la variable será el mismo que el del valor al que hace referencia(tipado dinámico). Por lo tanto su tipo puede cambiar en cualquier momento:

```
>>> var = 5
>>> print(type(var))
<class 'int'>

>>> var = "hola"
>>>print( type(var))
<class 'str'>
```

Hay que tener en cuenta que python distingue entre mayúsculas y minúsculas en el nombre de una variable, pero se recomienda usar sólo minúsculas.
Para crear una variable simplemente tenemos que utilizar un operador de asignación, el más utilizado = para que referencia un valor. Si queremos borrar la variable utilizamos la instrucción del.

```
>>> a = 5
>>> a
5

>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Podemos tener también variables que no tengan asignado ningún tipo de datos:

```
>>> a = None
>>> type(a)
<class 'NoneType'>
```

En python las constantes se declaran en un archivo(suele llamarme módulo) aparte, su nombre en mayúsculas. Para utilizar el módulo en nuestro archivo de código tenemos que importarlo (import). Una vez importado para acceder a las constantes

nombremodulo.nombre contante

6. Tipos de datos

Podemos concretar aún más los tipos de datos (o clases) de los objetos que manejamos en el lenguaje:

- Tipos númericos
  - Tipo entero (int)
  - Tipo real (float)
  - Tipo complejo (complex)
- Tipos booleanos (bool)
- Tipo de datos secuencia
  - Tipo lista (list)
  - Tipo tuplas (tuple)
  - Tipo rango (range)
- Tipo de datos cadenas de caracteres
  - Tipo cadena (str)
- Tipo de datos binarios
  - Tipo byte (bytes)
  - tipo bytearray (bytearray)
- Tipo de datos conjuntos
  - Tipo conjunto (set)
  - Tipo conjunto inmutable (frozenset)

- Tipo de datos iterador y generador (iter)
- Tipo de datos mapas o diccionario (dict)

Tipos Numéricos

Además de int y float, python admite otros tipos de números, como Decimal y Fraction. También tiene soporte incorporado para números complejos, y usa el sufijo i o j para indicar la parte imaginaria

La división siempre retorna un punto flotante. Para obtener un resultado entero puede usarse el operador //

Con python, es posible usar el operador ** para calcular potencias

Cadenas (clase str)

Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas sencillas ' o dobles “. Cada carácter tiene asociado un índice que permite acceder a él, empezamos en 0
También se pueden utilizar índices negativos para recorrer la cadena del final al principio.
El índice del último carácter de la cadena es -1.

```
>>> 'Python'[-1]
'n'
```

Para trabajar con subcadenas cadena[i:j:k]

Devuelve la subcadena desde el carácter con el índice i hasta el carácter anterior al índice j tomando caracteres cada k

```
>>> 'Python'[1:4]
'yth'

>>> 'Python'[1:1]
''
>>> 'Python'[2:]
'thon'

>>> 'Python'[:-2]
```

```
'Pyth'

>>> 'Python'[:]
'Python'

>>> 'Python'[0:6:2]
'Pto'
```

chr(i): Nos devuelve el carácter Unicode que representa el código i.

```
>>> chr(97)
'a'
>>> chr(1004)
'Ϭ'
```

ord(c): recibe un carácter c y devuelve el código unicode correspondiente.

```
>>> ord("a")
97
>>> ord("Ϭ")
1004
```

Conversión de tipos

- int(x): Convierte el valor a entero.
- float(x): Convierte el valor a float.
- complex(x): Convierte el valor a un complejo sin parte imaginaria.
- complex(x,y): Convierta el valor a un complejo, cuya parte real es x y la parte imaginaria y.

Los valores que se reciben también pueden ser cadenas de caracteres (str).

Ejemplos

```
>>> a=int(7.2)
>>> a
7
>>> type(a)
<class 'int'>

>>> a=int("345")
>>> a
345
>>> type(a)
<class 'int'>

>>> b=float(1)
>>> b
1.0
>>> type(b)
<class 'float'>

>>> b=float("1.234")
>>> b
1.234
>>> type(b)
<class 'float'>
```

Por último si queremos convertir una cadena a entero, la cadena debe estar formada por caracteres numéricos, sino es así, obtenemos un error:

```
a=int("123.3")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.3'
```

Función type()

La función type nos devuelve el tipo de dato de un objeto dado.

Por ejemplo:

```
>>> type(5)
<class 'int'>

>>> type(5.5)
<class 'float'>

>>> type([1,2])
<class 'list'>

>>> type(int)
<class 'type'>
```

Función isinstance()

Esta función devuelve True si el objeto indicado es del tipo indicado, en caso contrario devuelve False.

```
>>> isinstance(5,int)
True

>>> isinstance(5.5,float)
True

>>> isinstance(5,list)
```

```
False
```

Como hemos indicado anteriormente las variables en python no se declaran, se determina su tipo en tiempo de ejecución empleando una técnica que se llama **tipado dinámico**.

Cuando asignamos una variable, se crea una referencia (puntero) al objeto creado, en ese momento se determina el tipo de la variable. Por lo tanto cada vez que asignamos de nuevo la variable puede cambiar el tipo en tiempo de ejecución.

```
>>> var = 3
>>> type(var)
<class 'int'>
>>> var = "hola"
>>> type(var)
<class 'str'>
```

7. Operadores.

Podemos clasificarlos en varios tipos:

- Operadores aritméticos
- Operadores de cadenas
- Operadores de asignación
- Operadores de comparación
- Operadores lógicos
- Operadores a nivel de bit
- Operadores de pertenencia
- Operadores de identidad

Operadores de asignación

Me permiten asignar un valor a una variable, o mejor dicho: me permiten cambiar la referencia a un nuevo objeto.

El operador principal es `=`

```
>>> a = 7
>>> a
```

```
7
```

Otros operadores de asignación: `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`

Asignación múltiple

En python se permiten asignaciones múltiples de esta manera:

```
>>> a, b, c = 1, 2, "hola"
```

Operadores de identidad

- `is`: Devuelve True si dos variables u objetos están **referenciando** la misma posición de memoria. En caso contrario devuelve False. No es sólo comparar contenido (==)
- `is not`: Devuelve True si dos variables u objetos **no** están referenciando la misma posición de memoria. En caso contrario devuelve False.

Ejemplo

```
>>> a = 5
>>> b = a
>>> a is b
True

>>> b = b +1
>>> a is b
False

>>> b is 6
True
```

Operadores aritméticos

- `+`: Suma dos números
- `-`: Resta dos números
- `*`: Multiplica dos números
- `/`: Divide dos números, el resultado es `float`.
- `//`: División entera
- `%`: Módulo o resto de la división
- `**`: Potencia
- `+`, `-`: Operadores unarios positivo y negativo

Operadores a nivel de bit

- `x | y`: x OR y
- `x ^ y`: x XOR y
- `x & y`: a AND y
- `x << n`: Desplazamiento a la izquierda **n** bits.
- `x >> n`: Desplazamiento a la derecha **n** bits.
- `~x`: Devuelve los bits invertidos.

Operadores lógicos

Los operadores booleanos se utilizan para operar sobre expresiones booleanas y se suelen utilizar en las estructuras de control alternativas (if, while):

- `x or y`: Si x es falso entonces y, sino x. Este operados sólo evalua el segundo argumento si el primero es False.
- `x and y`: Si x es falso entonces x, sino y. Este operados sólo evalua el segundo argumento si el primero es True.
- `not x`: Si x es falso entonces True, sino False.

Funciones all() y any()

- all(iterador): Recibe un iterador, por ejemplo una lista, y devuelve True si todos los elementos son verdaderos o el iterador está vacío.
- any(iterador): Recibe un iterador, por ejemplo una lista, y devuelve True si alguno de sus elemento es verdadero, sino devuelve False.

Operadores de comparación

== != >= > <= <

Operadores de cadena

- +: concatena
- *: repite la cadena n veces

```
>>> c = "tres"
>>> c * 3
'trestrestres'
```

Dos o más cadenas literales (es decir, las encerradas entre comillas) una al lado de la otra se concatenan automáticamente.

```
>>>
>>> 'Py' 'thon'
'Python'
```

Esta característica es particularmente útil cuando quieres dividir cadenas largas:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Esto solo funciona con dos literales, no con variables ni expresiones

Esto mismo lo puedes hacer usando las triples comillas “””

Operadores de pertenencia

Un operador de pertenencia se emplea para identificar pertenencia en alguna secuencia (listas, strings, tuplas).

- In: devuelve True si el valor especificado se encuentra en la secuencia. En caso contrario devuelve False.
- not in: devuelve True si el valor especificado no se encuentra en la secuencia. En caso contrario devuelve False.

8. Objetos inmutables y mutables

Objetos inmutables

Python procura no consumir más memoria que la necesaria. Ciertos objetos son **inmutables**, es decir, no pueden modificar su valor. El número 2 es siempre el número 2. Es un objeto inmutable. Python procura almacenar en memoria una sola vez cada valor inmutable. Si dos o más variables contienen ese valor, sus referencias apuntan a la misma zona de memoria.

Ejemplo

Para comprobar esto, vamos a utilizar la función id, que nos devuelve el identificador de la variable o el objeto en memoria.

Veamos el siguiente código:

```
>>> a = 5
```

Podemos comprobar que a hace referencia al objeto 5.

```
>>> id(5)
10771648
>>> id(a)
10771648
```

Esto es muy distinto a otros lenguajes de programación, donde una variable ocupa un espacio de memoria que almacena un valor. Cuando asigno otro número a la variable estoy cambiando la referencia.

```
>>> a = 6
>>> id(6)
10771680
>>> id(a)
10771680
```

Las cadenas también son un objeto **inmutable**, que lo sean tiene efectos sobre las operaciones que podemos efectuar con ellas. La asignación a un elemento de una cadena, por ejemplo está prohibida:

```
>>> a = "Hola"
>>> a[0]="h"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

De los tipos de datos principales, hay que recordar que son inmutables los números, las cadenas o las tuplas.

Objetos mutables

El caso contrario lo tenemos por ejemplo en los objetos de tipo listas, en este caso las listas son mutables. Se puede modificar un elemento de una lista.

Ejemplo

```
>>> a = [1,2]
>>> b = a
>>> id(a)
140052934508488
>>> id(b)
140052934508488
```

Como anteriormente vemos que dos variables referencia a la misma lista en memoria. Pero aquí viene la diferencia, al poder ser modificada podemos encontrar situaciones como la siguiente:

```
>>> a[0] = 5
>>> b
[5, 2]
```

Cuando estudiamos las listas veremos este compartiendo de manera completa.

De los tipos de datos principales, hay que recordar que son mutables son las listas y los diccionarios.

9. Entrada y salida estándar

Función input

Nos permite leer por teclado información. Devuelve una cadena de caracteres y puede tener como argumento una cadena que se muestra en pantalla.

Ejemplos

```
>>> nombre=input("Nombre:")
Nombre:jose
>>> nombre
'jose'

>>> edad=int(input("Edad:"))
Edad:23
>>> edad
23
```

Función print

Nos permite escribir en la salida estándar. Podemos indicar varios datos a imprimir, que por defecto serán separados por un espacio (se puede indicar el separador) y por defecto se termina con un carácter salto de línea \n (también podemos indicar el carácter final). Podemos también imprimir varias cadenas de texto utilizando la concatenación.

Ejemplos

```
>>> print(1,2,3)
1 2 3
>>> print(1,2,3,sep="-")
1-2-3

>>> print(1,2,3,sep="-",end=".")
1-2-3.>>>

>>> print("Hola son las",6,"de la tarde")
Hola son las 6 de la tarde

>>> print("Hola son las "+str(6)+" de la tarde")
Hola son las 6 de la tarde
```

Formateando cadenas de caracteres

Existen dos formas de indicar el formato de impresión de las cadenas. En la documentación encontramos el estilo antiguo y el estilo nuevo.

Ejemplos del estilo antiguo

```
>>> print("%d %f %s" % (2.5,2.5,2.5))
2 2.500000 2.5

>>> print("%s %o %x"%(bin(31),31,31))
```

0b11111 37 1f

```
>>> print("El producto %s cantidad=%d precio=%.2f"%("cesta",23,13.456))
El producto cesta cantidad=23 precio=13.46
```

Función format()

Para utilizar el nuevo estilo en python3 tenemos una función format y un método format en la clase str

- c.format(valores): Devuelve la cadena c ras sustituir los valores de la secuencia valores en los marcadores de posición de c. Los marcadores de posición se indican mediante llaves {} en la cadena, y el reemplazo de los valores se puede realizar por posición, indicando en número de orden del valor dentro de las llaves, o por nombre, indicando el nombre del valor, siempre y cuando los valores se pasen con el formato nombre = valor.

```
>>> 'Un {} vale {} {}'.format('€', 1.12, '$')
'Un € vale 1.12 $'
>>> 'Un {2} vale {1} {0}'.format('€', 1.12, '$')
'Un $ vale 1.12 €'
>>> 'Un {moneda1} vale {cambio} {moneda2}'.format(moneda1 = '€', cambio = 1.12, moneda2 = '$')
'Un € vale 1.12 $'
```

Los marcadores de posición, a parte de indicar la posición de los valores de reemplazo, pueden indicar también el formato de estos. Para ello se utiliza la siguiente sintaxis:

- {:n} : Alinea el valor a la izquierda rellenando con espacios por la derecha hasta los n caracteres.
- {:>n} : Alinea el valor a la derecha rellenando con espacios por la izquierda hasta los n caracteres.
- {:^n}: Alinea el valor en el centro rellenando con espacios por la izquierda y por la derecha hasta lo n caracteres.
- {:nd}: Formatea el valor como un número entero con n caracteres rellenan do con espacios blancos por la izquierda.
- {:n.mf}: Formatea el valor como un número real con un tamaño de n caracteres (incluido el separador de decimales) y m cifras decimales, rellenando con espacios blancos por la izquierda.

```
>>> 'Hoy es {:^10}, mañana {:10} y pasado {:>10}'.format('lunes', 'martes', 'miércoles')
'Hoy es   lunes   , mañana martes     y pasado  miércoles'
```

```
>>> 'Cantidad {:5d}'.format(12)'
'Cantidad    12'
>>> 'Pi vale {:8.4f}'.format(3.141592)
'Pi vale   3.1416'
```

10. Funciones predefinidas

Numéricas

- abs(x): Devuelve al valor absoluto de un número.
- divmod(x,y): Toma como parámetro dos números, y devuelve una tubla con dos valores, la división entera, y el módulo o resto de la división.
- hex(x): Devuelve una cadena con la representación hexadecimal del número que recibe como parámetro.
- oct(x): Devuelve una cadena con la representación octal del número que recibe como parámetro.
- bin(x): Devuelve una cadena con la representación binaria del número que recibe como parámetro.
- pow(x,y): Devuelve la potencia de la base x elevado al exponente y. Es similar al operador **`.
- round(x,[y]): Devuelve un número real (float) que es el redondeo del número recibido como parámetro, podemos indicar un parámetro opcional que indica el número de decimales en el redondeo.

Ejemplos

```
>>> pow(2,3)
8

>>> round(7.567,1)
7.6
```

Funciones de cadenas

- len(c) : Devuelve el número de caracteres de la cadena c.

- min(c): Devuelve el carácter menor de la cadena c.
- max(c): Devuelve el carácter mayor de la cadena c.
- c.upper(): Devuelve la cadena con los mismos caracteres que la cadena c pero en mayúsculas.
- c.lower(): Devuelve la cadena con los mismos caracteres que la cadena c pero en minúsculas.
- c.title(): Devuelve la cadena con los mismos caracteres que la cadena c con el primer carácter en mayúsculas y el resto en minúsculas.
- c.split(delimitador): Devuelve la lista formada por las subcadenas que resultan de partir la cadena c usando como delimitador la cadena delimitador. Si no se especifica el delimitador utiliza por defecto el espacio en blanco.

```
>>> 'A,B,C'.split(',')
['A', 'B', 'C']
```