Introducción a la programación en Python

[6.1] ¿Cómo estudiar este tema?

[65.2] Estructura de un programa

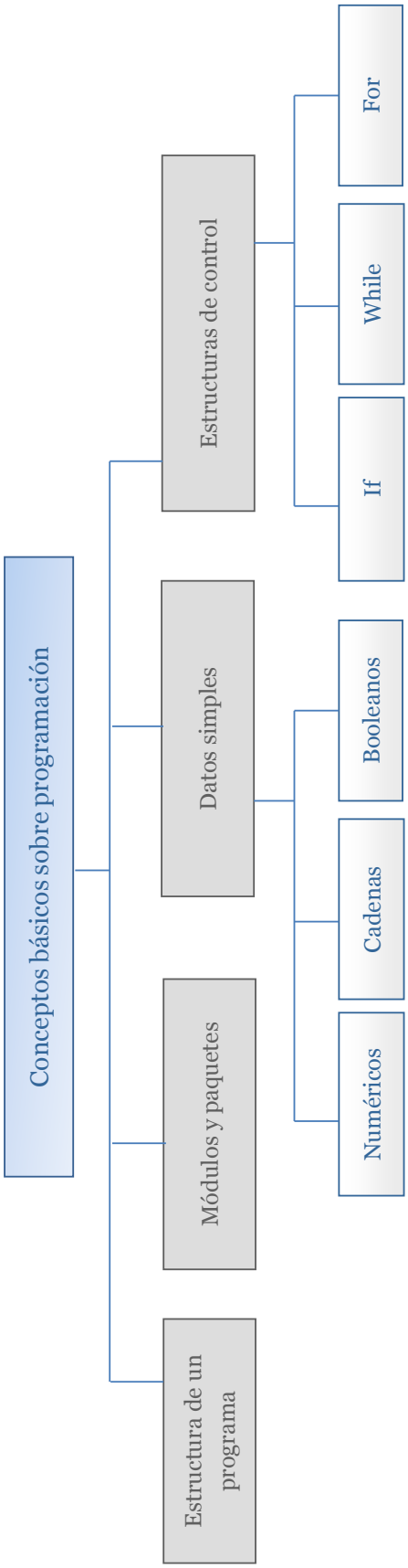
[6.3] Módulos y paquetes

[6.4] Tipos de datos simples

[6.5] Estructuras de control

TEMA 6

Esquema


Conceptos básicos sobre programación
Estructuras de control
For
While
If
Datos simples
Booleanos
Cadenas
Numéricos
Módulos y paquetes
Estructura de un programa

Ideas clave

6.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás comprender las **Ideas clave** expuestas en este documento, que se complementan con lecturas y otros recursos para que puedas ampliar los conocimientos sobre el mismo.

En este tema se explica **cómo estructurar un programa en Python**, así como los primeros pasos para empezar a utilizar dicho lenguaje. En este tema lo esencial es

- **Estructura de un programa en Python.**
- **Módulos** de Python: qué son, cómo se utilizan.
- **Datos simples**: cómo se definen y principales operaciones.
- **Estructuras de control en Python.**

Para una compresión óptima del tema, se recomienda tener ya instalado Python e ir ejecutando las sentencias que se muestran como ejemplo en un fichero `.py` o en el intérprete (esto se explica en la actividad no puntuable).

6.2. Estructura de un programa en Python

Cada lenguaje de programación tiene una estructura determinada que debe respetarse a la hora de hacer un programa. Algunos lenguajes, como C/C++ o Pascal tienen una estructura muy rígida, que exigen que se comience con identificadores o palabras reservadas. Sin embargo, **Python es muy flexible** en este sentido y una sola línea de código puede constituir un programa.

Veamos un ejemplo de programa en Python:

```
def suma(a, b):
    suma = (a + b) * 3    #Devuelve la suma multiplicada por 3
    return(suma)
```

Este programa recibe dos sumandos como entrada y devuelve su suma multiplicada por tres. A pesar de su sencillez, nos va a permitir describir los aspectos esenciales del código en Python.

En primer lugar, no hay una palabra (como `falg` en pseudocódigo) o símbolo (como llaves o corchetes) que indique el fin del algoritmo.

Por esto, en Python es esencial la **indentación** (y no solo deseable) ya que es así como se marca no solo el final de un algoritmo sino el de cualquier otro elemento, como una estructura selectiva, por ejemplo.

Es importante hacer una **indentación consistente**. En principio, se puede indentar utilizando espacios en blanco o el tabulador, pero no deben mezclase ambos en un mismo código, ya que puede ocurrir en este caso que el editor en que se escribe el programa no vea la misma indentación que Python y devuelva en consiguiente error. En este curso, utilizaremos la forma de indentación recomendada: 4 espacios en blanco.

Otras diferencias con respecto al pseudocódigo que se ha visto es que en Python **no es necesario declarar variables** (a diferencia de lo que ocurre en muchos otros lenguajes). Es decir, no necesito especificar al principio del algoritmo qué variables van a utilizarse y de qué tipo son. De hecho, en este sentido la flexibilidad de Python es absoluta: una variable puede ser de tipo entero y a continuación sobrescribirla como carácter, por ejemplo.

El último aspecto reseñable es el uso del símbolo `#` para hacer comentarios. Si se escribe el símbolo `#`, Python ignorará el texto que esté escrito a continuación y no se ejecutará.

Los comentarios son de gran ayuda en el desarrollo de programas, ya que facilita la búsqueda de errores y las actualizaciones de código. Pero también hay que tener cuidado con el empleo de los comentarios y no utilizarlos donde no sean necesarios. El comentario del ejemplo es un tipo de comentario que no debe hacerse, ya que se limita a duplicar la información (traduce el lenguaje de Python al lenguaje natural). Los comentarios deben reservarse para explicar aquello que no es evidente.

Ejecución de un programa

Los programas Python se pueden escribir en cualquier editor de texto plano (notepad o TextEdit, por ejemplo) y se guardan por lo general con una extensión `.py`. Una vez creados, pueden ejecutarse dentro de una consola o del intérprete de Python (esto último solo es útil si el código es breve). La consola en Windows también recibe el nombre de «Símbolo del sistema» y en Mac OS X o Linux «Terminal».

Para ejecutar código con el intérprete de Python simplemente hay que escribirlo tras el símbolo `>>>>`. Cuando haya una línea en blanco se ejecutará el código introducido. En cambio, si el código se ejecuta desde una consola, las líneas en blanco se ignoran.

Para familiarizarse con la sintaxis de Python, es recomendable escribir las operaciones y ejecutar las sentencias, que se explican en el resto del tema, en el intérprete de Python.

6.3. Módulos y paquetes

Supongamos que estamos trabajando en el intérprete de Python, implementando algoritmos (funciones) y definiendo variables. En el momento en que salgamos del intérprete, todo el trabajo que hayamos realizado se perderá. Para evitar tener que repetir todas las instrucciones se puede crear un **script**, esto es, un fichero de texto en el que se recojan todas las instrucciones y definiciones que son necesarias para trabajar.

Esto puede ser una solución si el script resultante no es demasiado grande, porque, ¿qué ocurre si se quieren utilizar las funciones de otras formas? ¿Hay que editar cada vez el script original y copiar de allí los fragmentos que nos interesa rescatar? Esto puede ser realmente incómodo, sobre todo si se manejan muchas funciones distintas y el script resultante es de gran tamaño.

La solución que da Python a esto es la creación de módulos. Un **módulo** o **biblioteca** es un conjunto de variables y funciones (entre otras cosas) que, una vez importadas, pueden ser utilizadas desde el módulo principal o desde otros módulos. Es decir, un módulo en Python sirve para poder **organizar código**.

Los módulos de Python se almacenan en un fichero de extensión `.py`.

La manera de importar módulos es muy sencilla, basta con ejecutar:

```
>>>> import modulo
```

También se pueden importar varios módulos en una misma línea:

```
>>>> import modulo1, moduo2, modulo3
```

Si en el módulo `modulo` está definida la función `funcion`, para acceder a ella se debe ejecutar:

```
>>>> modulo.funcion(argumentos)
```

Es decir, se debe anteponer el nombre del módulo en que se encuentran (o el **espacio de nombres**). Esto se hace para evitar confusiones entre objetos de distintos módulos que tengan el mismo nombre. No obstante, puede utilizarse la siguiente instrucción para evitar tener que especificar el espacio de nombres cada vez que se llame a una función determinada:

```
>>>> from modulo import objeto
```

Así, si se quiere llamar ahora a la función `funcion` basta con ejecutar:

```
>>>> funcion(argumentos)
```

Si se quiere hacer esto mismo con todos los objetos de un módulo basta con hacer un *wildcar*, es decir, ejecutar:

```
>>>> from modulo import *
```

Aunque hay que tener cuidado con esta práctica.

Biblioteca estándar

No todos los módulos tienen que ser creados por el usuario. De hecho, Python tiene sus propios módulos en la **biblioteca estándar**.

Esto es lo que se conoce como filosofía de *pilas incluidas,* ya que liberan al usuario de la tarea de programarlas. Hay decenas de módulos, que no pueden estudiarse en este curso. En este subapartado se mencionan algunos de los más importantes y, a lo largo del curso, se irán añadiendo nuevos módulos conforme se vayan necesitando.

El módulo `os` permite trabajar con el sistema operativo. Tiene implementadas funciones que permiten realizar tareas como eliminar archivos, cambiar permisos a un archivo, obtener la lista de contenido del directorio, etc.

Este módulo proporciona un buen ejemplo de por qué hay que ser muy cuidadoso con el *wildcar*. Si se ejecuta:

```
>>>> from os import *
```

la función `os.open()` ocultará la función interna `open()`, que opera a nivel más alto.

Para trabajar con el sistema operativo a más alto nivel está implementado el módulo `shutil`, que permite copiar ficheros, eliminar directorios enteros u obtener estadísticas del uso del disco, entre otras funciones.

El módulo `sys` se utiliza para acceder a variables que utiliza el intérprete así como a funciones que interactúan fuertemente con el intérprete. Tiene implementadas funciones para forzar la salida del intérprete, obtener la versión de Python o mostrar información de los módulos, por ejemplo.

Paquetes

Igual que los módulos permiten tener el código ordenado, **los paquetes permiten organizar los módulos**. Así, un paquete es un directorio que agrupa varios módulos que están relacionados.

Para que Python reconozca una carpeta como paquete tiene que incluir un fichero `__init__.py`, que puede estar vacío.

Dentro de un paquete puede haber subpaquetes, en sus correspondientes directorios y con su fichero `__init__.py`.

Los paquetes también se importan con los comandos `import` y `from … import`. Así

```
>>>> import modulo
>>>> import paquete.modulo
>>>> import paquete.subpaquete.modulo
```

La primera sentencia es para importar un módulo que no pertenece a ningún paquete, la segunda importa un módulo que está en un paquete y la tercera, un módulo que está en un subpaquete. Análogamente,

```
>>>> from paquete.modulo import *
```

Importa todos los objetos del módulo `modulo` que está en el paquete `paquete`.

6.4. Tipos de datos

A continuación veremos en detalle cómo se definen los distintos tipos de datos en Python y cómo se realizan las operaciones más comunes. Conviene recordar las reglas, que se vieron en el tema 3, para dar nombres a las distintas variables.

Enteros (integer)

Son **números**, positivos o negativos, **sin parte decimal**, cuyo tamaño máximo está determinado por la memoria de la máquina. Si definimos, por ejemplo, los enteros `x` e `y`,

```
>>>> x = 32
>>>> y = -3
```

Pueden realizarse operaciones numéricas (Tabla 5.1)

Sintaxis	Descripción
`x + y`	Suma `x` e `y`
`x - y`	Resta `y` a `x`
`x * y`	Multiplica `x` por `y`
`x / y`	Divide `x` entre `y`. Siempre produce un valor `float`
`x // y`	Divide `x` entre `y` y trunca el resultado. Siempre produce un valor `int`
`x % y`	Devuelve el resto de dividir `x` entre `y`.
`x ** y`	Eleva `x` a `y`
`-x`	Cambia `x` de signo, si x es distinto de cero

Tabla 5.1: Operaciones numéricas en Python

Los operadores numéricos (`+, -, /, //, %` y `**`) tienen versiones de asignación aumentada (`+=, -=, /=, //=, %=` y `**=`). Por ejemplo, escribir `x += y` equivale a `x = x + y`:

```
>>>> x += y
>>>> x
31
```

También puede aplicarse las siguientes funciones (Tabla 5.2)

Sintaxis	Descripción
`abs(x)`	Devuelve el valor absoluto de `x`
`divmod(x, y)`	Devuelve el cociente y el resto de la división `x` entre `y`
`pow(x, y)`	Eleva `x` a `y`. Hace lo mismo que el operador **
`round(x)`	Devuelve un valor `int`, correspondiente al redondeo de `x`
`round(x, n)`	Devuleve un valor `float`, que es el valor `x` con `n` decimales

Tabla 5.2: Funciones numéricas en Python

Para definir una variable (también podemos decir objeto) de tipo entera en Python pueden ejecutarse:

```
>>>> x = 32
>>>> x = int(32)
>>>> x = int()
```

En el primer caso, se utiliza el símbolo `=` para asignar el número `32` a la variable `x`. En el segundo caso, estamos utilizando la función `int()`. Para el caso de otro tipo de datos numéricos va a ser esencial llamar a una función, aunque para los datos tipo entero no es necesario. En el tercer caso, se crea un entero con valor 0.

Como en informática también se utilizan mucho los sistemas de numeración binario, octal y hexadecimal, existen funciones de conversión de enteros (Tabla 5.3)

Sintaxis	Descripción
`bin(i)`	Devuelve la representación binaria de `i`
`hex(i)`	Devuelve la representación hexadecimal de `i`
`int(x)`	Convierte `x` en entero. Si `x` es un número de punto flotante, lo trunca
`int(s)`	Convierte la cadena `s` en entero
`oct(i)`	Devuelve la representación octal de `i`

Tabla 5.3: Funciones de conversión en Python

Números de punto flotante (float)

Los **números de tipo flotante** o real son **números con parte decimal**. Este tipo de datos ofrece una gran velocidad de cálculo pero, como se vio en el tema 3, no es del todo precisa su representación. La exactitud de los valores `float` de Python puede variar de una máquina a otra, aunque por lo general los primeros 15 dígitos son fiables.

Pueden utilizarse con números de punto flotante todos los operadores y funciones de la Tablas 4.1 y 4.2, además de los operadores de asignación aumentada.

Hay varias formas de transformar un número de punto flotante en un entero (Tabla 5.4). Para algunas de ellas hay que importar el módulo `math`.

Sintaxis	Descripción
`round(x)`	Devuelve el valor redondeado de `x`
`math.floor(x)`	Devuelve el valor redondeado hacia arriba de `x`
`math.ceil(x)`	Devuelve el valor truncado de `x`

Tabla 5.4: Funciones de conversión de `float` a `int`

Como se ha visto, si se realiza la asignación:

```
>>>> x = 32.0
```

`x` será almacenado como `float`, pero puede evaluarse si en realidad es o no entero: el método `float.is_integer()` devuelve `True` si la parte fraccional es cero. También pueden escribirse expresiones como:

```
>>>> x = float(2.3)
```

Existen gran cantidad de funciones que trabajan con datos tipo float en el módulo math. Algunas de ellas se muestran en la Tabla 5.5.

Sintaxis	Descripción
`math.copysign(x)`	Devuelve el signo de `x`
`math.factorial(x)`	Devuelve el factorial de `x`
`math.degrees(r)`	Convierte los radianes (r) en grados
`math.isinf(x)`	Devuelve `True` si `x` es $\pm\infty$
`math.isnan(x)`	Devuelve `True` si `x` es `NaN` (Not a Number)

Tabla 5.5: Funciones del módulo `math`

Decimales

La representación de los datos de punto flotante hace que no se pueda garantizar la exactitud de los números y, entre otras cosas, no deban usarse si se quieren hacer comparaciones fiables de igualdad.

Python tiene el **tipo de datos** `Decimal`, pertenecientes al módulo `decimal` que **garantizan la exactitud** de los mismos, eso sí, perdiendo velocidad de cálculo. Para trabajar con estos números hay que importar el módulo decimal y crearlos a través de la función `Decimal`, ya que este tipo no admite representación literal de datos.

```
>>>> import decimal
>>>> q = decimal.Decimal(1234)
>>>> r = decimal.Decimal("0.987654321")
>>>> q + r
Decimal('1234.987654321')
```

Nótese que se han introducido en `Decimal` números de dos tipos distintos: entero para `q` y cadena para `r`. No puede pasarse un `float` como argumento en un dato de tipo `Decimal`, ya que los decimales de un número de punto flotante se almacenan con poca exactitud.

Todos los operadores y funciones de las Tablas 4.1 y 4.2 pueden emplearse también para números de tipo decimal, así como la asignación aumentada.

Aunque operaciones como la división son más exactas con el formato de número decimal que con el de coma flotante, no debería haber discrepancias entre resultados antes del decimoquinto decimal, por lo que, en adelante, cuando se hable de números con parte decimal, se asumirá que son de tipo `float`.

Complejos

Estos números, muy interesantes en aplicaciones de matemáticas, física o ingeniería, no van a tener demasiadas aplicaciones en este curso. Por esta razón, nos limitaremos a definirlos y comentar las operaciones más básicas.

Los **números complejos** son una extensión de los números reales y **se representan de la forma z = a + bj**, donde a y b son números reales y j es la parte imaginaria (en matemáticas y física la parte imaginaria se denota con la letra i, pero en Python se sigue la notación que se usa en ingeniería). Por tanto, para definir el número complejo z = 3 + 2j en Python podría simplemente hay que escribir:

```
>>>> z = 3 + 2j
```

y para saber cuáles son las partes real e imaginaria de un número complejo se ejecuta:

```
>>>> z.real, z.imag
(3, 2)
```

Todos los operadores y funciones de las Tablas 4.1 y 4.2 pueden emplearse con números complejos, salvo `//, % divmod()` y `pow()`.

Booleanos

En Python existen **dos objetos booleanos**: **True** y **False**. Las maneras de realizar asignaciones booleanas son:

```
>>>> b = True
>>>> c = False
>>>> d = bool(1)
>>>> e = bool()
```

Como ya podrá imaginar el alumno, `bool(0)` devuelve `False` y `bool(1)` devuelve `True`.

No es infrecuente, de hecho, encontrar código en el que se emplea `1` en lugar de `True` y `0` en lugar de `False`, por ejemplo, para realizar las asignaciones aumentadas emplear:

```
>>>> i += True
```

en lugar de:

```
>>>> i += 1
```

En general, no es una práctica recomendable y puede llevar a error. Lo correcto es utilizar valores booleanos cuando se requiera un valor booleano y un valor entero cuando se requiera un valor entero.

Recordemos que las operaciones entre datos de tipo booleano son `and, or, not`. Para saber el resultado de una operación booleana hay que tener en cuenta las tablas de verdad que se dan en el tema 3

Cadenas (strings)

Las cadenas se representan con el tipo de datos `str` (del inglés, *string*). Hay distintas formas de crear una cadena. La primera es utilizando comillas, que pueden ser simples o dobles.

```
>>>> a = ‘ejemplo de cadena’
>>>> b = "ejemplo de cadena"
>>>> a == b
True
```

Las cadenas se consideran objetos **medidos**, por lo que se puede calcular su longitud con la función `len()`, que devuelve el número total de caracteres (cero si la cadena es vacía)

```
>>>> len(a)
17
```

Una característica importante de las cadenas es que son objetos **inmutables**, es decir que no se puede modificar ninguno de sus caracteres:

```
>>> a = "cadena"
>>> a[2] = "f"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

También se pueden crear cadenas con la función `str()`, que también convierte en cadena variables de otro tipo:

```
>>>> a = srt(‘ejemplo de cadena’)
>>>> b = str(1984)
```

Es más habitual utilizar las dobles comillas, aunque es indistinto cuáles usar en la mayoría de los casos. Sí conviene, por comodidad, elegir comillas simples si la cadena contiene comillas dobles y al revés.

```
>>>> a = ‘cadena con comillas "dobles"’
>>>> b = "cadena con comillas ‘simples’"
```

Si se quiere realizar un salto de línea, se puede usar el símbolo `\n`:

```
>>>> a = ‘cadena con \n salto de linea’

>>> print (a)

cadena con

 salto de línea
```

Otra opción para incluir comillas en una cadena de caracteres es introducir los símbolos `\’` y `\”`

```
>>>> a = ‘cadena con el mismo tipo de \’comillas\’’
>>>> b = ‘cadena con el mismo tipo de \"comillas\"’
```

También pueden usarse cadenas crudas (raw). Son cadenas encerradas entre comillas dobles y que van precedidas de una letra `r`. En este caso, todo lo que vaya entre comillas será un literal y no es necesario hacer saltos de línea. Es necesario importar el módulo `re`.

```
>>>> import re
>>>> a = re.compile(r" cadena con comillas ‘simples’, "dobles"")
```

Si se quiere escribir un literal muy largo en dos o más líneas puede hacerse de dos formas:

```
>>>> a = "este no es el mejor modo de unir cadenas" + \

    "porque el salto de linea es muy poco elegante"
>>>> b = ("este es el mejor modo " "de unir cadenas largas")
```

Los símbolos especiales pueden representarse consultando su código Unicode. Por ejemplo,

```
>>>> euros = "€ \u20AC \U000020AC"
>>>> inf = "\u221e"
>>>> raiz = "\u23B7"
```

- Comparación de cadenas:

  Las cadenas pueden compararse utilizando los operadores de comparación normales <, <=, ==, =!, > y >= (menor que, menor o igual que, igual a, distinto de, mayor que y mayor o igual que, respectivamente). Por ejemplo, sean

```
>>>> a = "cadena"
>>>> b = "Cadena"
>>>> c = "CADENA"
>>>> d = ‘cadena’
```

  Se pueden hacer las comparaciones,

```
>>>> a = d
True
>>>> a < b
False
>>>> c != d
True
>>>> b >= c
True
```

Como se vio en el tema 3, **los operadores comparan las cadenas** carácter a carácter, en concreto, **bit a bit**. Pueden presentarse algunos problemas a la hora de realizar estas comparaciones, que no son exclusivos de Python sino que afectan a todo lenguaje que trabaje con Unicode.

Por ejemplo, algunos caracteres tienen más de una representación Unicode. Por otra parte, el orden de algunos caracteres puede variar en función del idioma. Además, algunos caracteres especiales como flechas o símbolos matemáticos, no tienen posiciones ordenadas. Esto puede dar lugar a errores muy difíciles de detectar en la comparación de cadenas. Por tanto, como norma general, en Python no se presupone nada: utiliza la representación del byte en la memoria de las cadenas.

- Recorrido de cadenas:

Como se vio en el tema 3, se pueden extraer subcadenas de una cadena dada. Python utiliza el elemento `[]`. Este operador es muy flexible, ya que permite la extracción de elementos con varias sintaxis. Sea la cadena

```
>>>> a = 'SOYLENT GREEN'
```

Las posiciones del índice de la cadena `a` se muestran en la Figura 5.1

a[-13]	a[-12]	a[-11]	a[-10]	a[-9]	a[-8]	a[-7]	a[-6]	a[-5]	a[-4]	a[-3]	a[-2]	a[-1]
S	**O**	**Y**	**L**	**E**	**N**	**T**		**G**	**R**	**E**	**E**	**N**
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]

Figura 5.1: Posiciones de una cadena

Además de las posiciones habituales, con números positivos, pueden utilizarse números negativos. Resulta especialmente útil la posición `[-1]`, que siempre devolverá el último carácter de la cadena, con independencia de su longitud.

Las sintaxis válidas para el operador `[]` aplicado a la cadena `cadena` son:

```
>>>> cadena [inicio]
>>>> cadena [inicio:fin]
>>>> cadena [inicio:fin:paso]
```

Aplicándolo a la cadena `a`, resulta:

```
>>>> a[4]
‘e’
```

Selecciona el elemento a[4] (ver Figura 5.1)

```
>>>> a[3:8]
‘lent ‘
```

Selecciona los elementos comprendidos entre `a[3]` y `a[8]` (ver Figura 5.1). Si se omite el índice de inicio se toma por defecto el `0`

```
>>>> a[:7]
‘soylent’
```

Y si se omite el índice de fin, se toma por defecto el último elemento de la cadena,

```
>>>> a[7:]
‘ green’
```

También puede emplearse con índices negativos,

```
>>>> a[-5:]
‘green’
```

Si se quisiera añadir la subcadena “`nt ` “ a la cadena `a` para formar ‘`soylent nt green`’

```
>>>> a = a[:8] + “nt “ + a[8:]
```

Si nos fijamos, la subcadena “`nt ` “ forma parte de la cadena original, por lo que también puede escribirse,

```
>>>> a = a[:8] + a[5:8] + a[8:]
```

Para recorrer toda la cadena `[0:13]` y devolver un elemento de cada dos,

```
>>>> a[0:13:2]
‘syetgen’
```

Como en el caso anterior, puede omitirse los índices de inicio y fin, por lo que la sentencia anterior es equivalente a

```
>>>> a[::2]
‘syetgen’
```

También puede emplearse índices negativos. Si el índice paso es negativo, se recorre la cadena de derecha a izquierda.

```
>>>> a[::-1]
'neerg tnelyos'
```

- Operaciones con cadenas:

Con las cadenas se pueden realizar, aparte de la operación `+` que ya se ha mencionado, el producto * y sus correspondientes asignaciones aumentadas:

```
>>>> a = "ja "
>>>> a*3
‘ja ja ja ’
```

Si se quiere

```
>>>> a += "jaa"
>>>> a
‘ja jaa’
```

Además, el método de cadena `s` permite realizar gran cantidad de operaciones con cadenas sin necesidad de tener que programarlas. Algunas de ellas se muestran en las Tablas 5.6 y 5.7, aplicadas a una cadena a.

Sintaxis	Descripción
`a.capitalize`	Devuelve una copia de la cadena con la primera letra en mayúsculas
`a.count(t, incio, fin)`	Devuelve la cantidad de ocurrencias de la subcadena `t` en `inicio:fin`
`a.find(t, inicio, fin)`	Devuelve la posición de `t` en `a` y `-1` si `t` no está en `a`
`a.index(t, inicio, fin)`	Devuelve la posición de `t` en `a` y una excepción `ValueError` si `t` no está en `a`
`a.isalnum()`	Devuelve `True` si `a` es no vacía y todos sus caracteres son alfanuméricos
`a.isalpha()`	Devuelve `True` si `a` es no vacía y todos sus caracteres son alfabéticos
`a.isnumeric()`	Devuelve `True` si `a` es no vacía y todos sus caracteres son dígitos o fracciones
`a.isdigit()`	Devuelve `True` si `a` es no vacía y todos sus caracteres son dígitos ASCII

Tabla 5.6: Algunos métodos de cadena (I)

Sintaxis	Descripción
`a.format(…)`	Devuelve una copia de `a` formateada según los argumentos que se le hayan pasado
`a.join(b)`	Devuelve la concatenación de cada elemento de `b` con un carácter de `a` entre cada uno
`a.just(long, char)`	Devuelve una copia de `a` alineada a la izquierda y con tantos caracteres char como haga falta para tener un total de long caracteres
`a.lower()`	Devuelve una copia de `a` en minúsculas
`a.upper()`	Devuelve una copia de `a` en mayúsculas
`a.replace(t, u, n)`	Devuelve todas las ocurrencias de `t` (hasta un máximo de n) reemplazadas por `u`
`a.strip(chars)`	Devuelve una copia de `a` sin los espacios en blanco del principio y del final
`a.title()`	Devuelve una copia de `a` con la primera letra de cada palabra en mayúsculas y el resto en minúsculas
`a.zfill(n)`	Devuelve una copia de `a` Si su longitud es menor que `b`, se rellena con ceros

Tabla 5.7: Algunos métodos de cadena (II)

Se recomienda al alumno que realice pruebas a modo de ejemplo para familiarizarse con las operaciones con cadenas y sus métodos así como con el entorno de Python. A modo de ejemplo, veremos el uso de algunos métodos, que son menos intuitivos.

Por ejemplo el método `.join()` que en principio parece tener poca utilidad

```
>>>> a = "casa"
>>>> b = "123"
>>>> a.join(b)
'1casa2casa3'
```

Permite concatenar las cadenas de una lista de modo muy sencillo (las listas se verán en detalle en el tema 8),

```
>>>> grado = ["Diseño", "de", "Programación"]
>>>> " ".join(grado)
'Diseño de Programación'
```

En este ejemplo también puede verse que no hace falta tener definida de antemano la cadena: se ha utilizado directamente `" "` con `.join(grado)`. Evidentemente, se pueden unir todos los elementos con el símbolo que se desee

```
>>>> "*-*".join(grado)
'Diseño*-*de*-*Programación'
```

Una manera muy útil de dar formato a las cadenas es con el método `.format()`. Supongamos que se quieren introducir determinados datos en un texto explicativo, por ejemplo,

```
>>>> a = "La asignatura {0} se cursó en el año {1}"
>>>> a.format("Fundamentos de programación", 2014)
'La asignatura Fundamentos de programación se cursó en el año 2014'
```

Como puede verse se ha sustituido `{0}` por la cadena `Fundamentos de programación` y `{1}` por la cadena `2014`.

Por supuesto pueden incluirse paréntesis con el método `.format()`

```
>>> a = "{{{0}}} {1}!"
>>> a.format("Yo estoy entre paréntesis", "pero yo no")
'{Yo estoy entre paréntesis} pero yo no!'
```

También puede emplearse para concatenar una cadena y un número (si se intenta hacer con el operador + devuelve error). Por ejemplo,

```
>>> a = "Juntar una {0} con un número como {1}"
>>> a.format("cadena", 34)
'Juntar una cadena con un número como 34'
```

También se puede dar formato a la cadena con argumentos de palabra clave

```
>>> a = "{quién} compró {cuántas} entradas"
>>> a.format(cuántas=6, quién = "Ella")
'Ella compró 6 entradas'
```

Esta forma hace que no tengamos porqué poner los argumentos en el mismo orden en que queramos emplearlos.

5.5. Estructuras de control

Vamos a ver ahora cómo se implementan las estructuras vistas en el tema 4, es decir, vamos a dar el salto del pseudocódigo al lenguaje Python. Es importante recordar que la indentación (es decir, el uso de espacios que marca los distintos bloques de código) es esencial en Python.

Estructuras selectivas

La sintaxis de la estructura **`si-entonces`** en Python es:

```
if <condición>:
    <acciones>
```

Es un error muy común al principio olvidar los dos puntos después de la condición (que en este caso, equivaldrían al entonces). También hay que señalar que no hay una instrucción que marque el final del bloque de acciones dentro de la estructura **si-entonces**, se sabe por la indentación del código.

Ejemplo 5.1. Se quiere almacenar en una variable `calificacion` si un alumno ha aprobado un examen o no. Si la nota es mayor o igual que cinco se almacenará "`Apto`" y si es menor, "`No apto`".

```
calificacion = "Apto"

if (nota < 5):
    calificacion = "No apto"
```

La sintaxis de la estructura selectiva doble en Python es,

```
if <condición>:
    <acciones 1>
else:
    <acciones 2>
```

Utilizando esta estructura, es posible reescribir el código anterior en una sola línea.

Ejemplo 5.2. Escribir la estructura anterior con una estructura selectiva doble.

La estructura puede escribirse de dos formas, una muy similar al pseudocódigo,

```
if (nota < 5):
    calificacion = "No apto"
else:
    calificacion = "Apto"
```

Y otra que solo precisa de una línea,

```
calificacion = "Apto" if (nota > 5) else "No apto"
```

El uso de los paréntesis en este caso es opcional. En general, su uso es recomendable porque no solo ayuda a comprender el código, sino porque no usarlo puede inducir a errores.

Supongamos que a la nota del alumno se le quiere sumar un punto si ha entregado un trabajo (o sea, si la variable `trabajo` es `True`) y nada si no lo ha hecho. El código sin paréntesis

```
nota = nota + 1 if trabajo else 0
```

funciona perfectamente si el alumno ha entregado el trabajo, pero si no lo ha hecho, en lugar de sumar cero a su nota, actualizará su valor a cero. Por tanto, la manera correcta de escribirlo es,

```
nota = nota + (1 if trabajo else 0)
```

Este tipo de errores son complicados de detectar, especialmente en un código largo. Por eso es importante adquirir metodologías de trabajo que los eviten, en la medida de lo posible.

Veamos por último cómo se implementa la estructura selectiva múltiple,

```
if <condición 1>:
    <acciones 1>
elif <condición 2>:
    <acciones 2>
…
elif <condición n>:
    <acciones n>
else:
    <acciones n + 1>
```

La cláusula `elif` equivale al **si-entonces** del pseudocódigo. La diferencia fundamental está en el indentado, que en Python no hace falta que muestre las sucesivas anidaciones.

Ejemplo 5.3. Se dan clases de tenis a tres grupos de alumnos: menores de 14 años, entre 15 y 25 años y mayores de 25 años. Se quiere hacer un código que en función de la edad coloque al alumno en el grupo que corresponda.

```
if (edad < 15):
    grupo = "Pequeños"
elif (edad >=25):
    grupo = "Adultos"
else:
    grupo = "Jóvenes"
```

Estructuras repetitivas

De todas las estructuras repetitivas del tema 4, en Python solo vamos a encontrarnos con dos: **estructura mientras** y **estructura desde**.

- Estructura mientras (while)

  La sintaxis básica de la estructura mientras es,

```
while <condición>:
    <acciones>
```

  Como ocurre con las estructuras selectivas, el inicio de las acciones comienza tras los dos puntos y el fin del bloque de código se marca con la indentación.

  Ejemplo 5.4. Imprimir los 12 primeros múltiplos de 8.

```
i = 1
while i <= 12:
    print(i*8)
    i = i + 1
```

  La sintaxis completa de la estructura mientras es,

```
while <condición>:
    <acciones 1>
else:
    <acciones 2>
```

La cláusula `else`, por tanto, es opcional. Si la `<condición>` es verdadera, se ejecutarán las `<acciones 1>`. Cuando sea `False` se pasará siempre al `else`, a no ser que se encuentre con una sentencia de salto `(break)`. Otras sentencias que evitan la cláusula `else` son `return` y las excepciones, que no se ven en este curso.

Ejemplo 5.5. Buscar en una cadena el carácter `"h"`. Si el carácter está en la cadena, devolver la posición en que se encuentra. Si `"h"` no está en la lista, devolver `-1`.

```
indice = 0
while indice < len(cadena):
    if cadena[indice] == "h":
        break
    indice += 1
else:
    indice = -1
```

- Estructura desde (for)

La sintaxis básica de la estructura desde o bucle for es

```
for <condición>:
    <acciones>
```

Que es completamente análoga a la estructura del bucle while.

Ejemplo 5.6. Imprimir los 12 primeros múltiplos de 8, utilizando un bucle for.

```
for i in range(1, 13):
    print(i*8)
```

Tal y como ocurre en el bucle while, la sintaxis completa de la estructura desde es,

```
for <condición>:
    <acciones 1>
else:
    <acciones 2>
```

Ejemplo 5.5. Buscar en una cadena el carácter `"h"`. Si el carácter está en la cadena, devolver la posición en que se encuentra. Si `"h"` no está en la lista, devolver `-1`. Emplear un bucle for.

```
indice = 0

for indice in range(len(cadena)):
    if cadena[indice] == "break":
        break
else:
    indice = -1
```

Lo + recomendado

No dejes de leer…

Errores comunes en Python

Lutz, M. (2004). *When Pythins Attack Common Mistakes of Python Programmers*. Recuperado de http://www.onlamp.com

En el enlace se explican algunos de los errores más comunes cuando se empieza a programar en Python (para este curso, basta con leer la primera página).

Accede al documento a través del aula virtual o desde la siguiente dirección web: http://www.onlamp.com/pub/a/python/2004/02/05/learn_python.html?page=1

Operadores lógicos en Pyhton

Montero, J. (2012). Python: Los operadores lógicos y sus tablas de verdad. Recuperado de http://elclubdelautodidacta.es

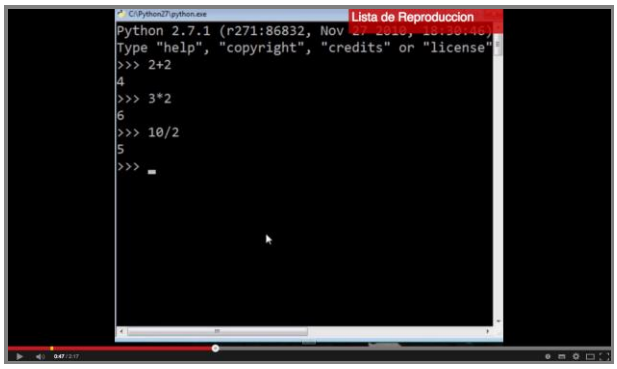
En el siguiente enlace se explica cómo definir los distintos operadores lógicos en Python y cómo comprobar las tablas de verdad de distintas operaciones lógicas.

Accede al documento desde el aula virtual o a través de la siguiente dirección web: http://elclubdelautodidacta.es/wp/2012/11/python-los-operadores-logicos-y-sus-tablas-de-verdad/

No dejes de ver...

Iniciación Python

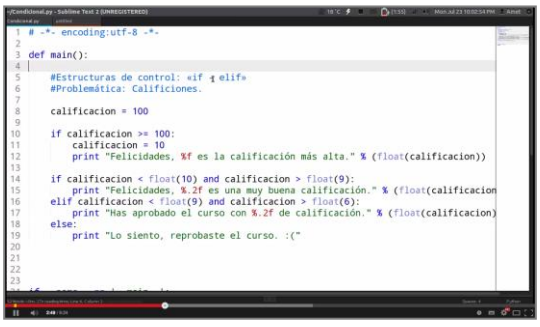
En los vídeos que se muestran a continuación se explica cómo usar el intérprete de Python.



Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=WoT65hvqGws

Estructuras de control if-elif

En los vídeos que se muestran a continuación se explica la sintaxis de las sentencias selectivas en Python.



Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=WoT65hvqGws

+ Información

A fondo

Python Loops

Tutorial para repasar las estructuras de control en Python.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
http://www.tutorialspoint.com/python/python_loops.htm

Python module index

Lista de todos los módulos Python, con su correspondiente enlace para ver en detalle todas las funciones implementadas.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
https://docs.python.org/3/library/aifc.html#module-aifc

Python package index

Lista de todos los paquetes Python, con su correspondiente enlace para ver en detalle todas las funciones implementadas.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
https://pypi.python.org/pypi/

Enlaces relacionados

Python

Python es un lenguaje de interpretado, gratuito y de código abierto.

Accede a la página web a través del aula virtual o desde la siguiente dirección:
https://www.python.org/
Tutoriales Python:
http://docs.python.org.ar/tutorial/pdfs/TutorialPython3.pdf

Bibliografía

Fernández, A. (2012). *Python 3 al descubierto*. Madrid: Ed. RC Libros.

Joyanes, L. (2008). *Fundamentos de la programación. Algoritmos y Estructura de Datos (4ª Ed)*. Madrid: McGraw-Hill.

López, J. & Quero, E. (2000). *Fundamentos de programación*. Madrid: Ed. Paraninfo.

Università di Napoli Federico II (2001). *The 68000's Instruction Set*. Recuperado de: http://www.grid.unina.it/

Valls, J. M. & Camacho, C. (2004). *Programación estructurada y algoritmos en PASCAL*. Madrid: Pearson Educación.

Von Neumann, J. (1945). *First Draft of a Report on the EDVAC*. Pennsylvania. Moore School of Electrical Engineering University of Pennsylvania.

Actividades

Trabajo: Escribir tres algoritmos

Metodología

Escribir los algoritmos correspondientes a los siguientes problemas en Python. Verificar que el código es correcto utilizando ejemplos.

- Se quiere buscar el carácter `"a"` dentro de una cadena y escribir, en una cadena nueva, el resto de caracteres, desde la primera `"a"` hasta el final (por ejemplo, en la cadena `"zapato"` se escribiría `"apato"`). Si `"a"` no está en la cadena, crear una cadena vacía.

- Comprobar si una variable numérica es de tipo real (flotante) o no. En caso de que sea entera o decimal, convertirla a real y si es complejo imprimir (`"No se puede transformar a real un numero complejo"`).

- Dadas tres cadenas, `cad1`, `cad2` y `cad3`, escribir un algoritmo que imprima la menor de las tres.

Criterios de evaluación

- Se evaluará el diseño y la claridad de los algoritmos, así como la utilidad de los ejemplos.

Extensión máxima: 5 páginas (1 página de portada, 1 página de índice, 1 página para explicar las estructura del programa, 3 páginas para escribir el código y los ejemplos, como máximo), fuente Georgia 11 e interlineado 1,5.

Instalar Python y ejecutar un programa de prueba

- Instalar una versión 3 de Python desde la web.
- Ejecutar un programa de prueba, para ello:
  - Crear la carpeta FundPy en el directorio ($ruta) que se desee
  - Abrir un editor de texto plano (notepad en Windows, textedit en mac) y escribir en él,

```
#!/urs/bin/env python3
print(“Hello world!”)
```

  - Guardar como hello.py (Cuidado con la extensión, guardar como texto plano)
  - Abrir un cmd y ejecutar (en Windows)

```
C:\>cd c:\$ruta\FundPy
C:\ruta\FundPy\>C:\Python30\python.exe hello.py
```

O, en sistemas Unix o Mac OS X

```
~ user$ cd $ruta\FundPy
~ user\ruta\FundPy $ python3 hello.py
```

Test

1. La indentación en Python:

A. Es una buena práctica, altamente recomendable.

B. Es esencial en la estructura de un programa.

C. Debe practicarse en general, aunque hay situaciones en que no se recomienda.

2. Los comentarios en el código:

A. Facilitan la comprensión del mismo cuando duplican la información.

B. No están aconsejados en programas con un gran número de líneas.

C. Deben dar información que no es evidente.

3. Un fichero que recoge un conjunto de variables y funciones que permite organizar el código en Python recibe el nombre de:

A. Módulo.

B. Programa.

C. Script.

4. ¿Con qué relacionarías la filosofía de pilas incluidas de Python?

A. Con la biblioteca estándar.

B. Con el *wildcar*.

C. Con el fichero `__init__.py`

5. Si `x = 3`, `x+= 5` es igual a:

A. 5

B. (3, 5)

C. 8

6. `x = 44.0` es una variable de tipo:

A. Float (real).

B. Decimal.

C. Entero (int).

7. `a = 'ejemplo'`. ¿Qué devuelve `a[2, 5]`?

A. emp

B. jemp

C. ep

8. La sentencia `else:`

A. Se puede emplear en las estructuras repetitivas.

B. Es obligatorio emplearlo en las estructuras selectivas.

C. A y B son incorrectas.

9. El código:

```
for i in range(1, 21):
    print(i)
```

A. Imprime los 20 primeros números.

B. Imprime los 21 primeros números.

C. Imprime los números 1 y 21.

10. Señala el código correcto:

A.

```
if (a < 4):
num = 2
elif (a >=7):
num = 5
else:
num = 10
```

B.

```
if (a < 4)
    num = 2
elif (a >=7)
    num = 5
else:
    num = 10
```

C.

```
if a < 4:
    num = 2
elif a >=7:
    num = 5
else:
    num = 10
```