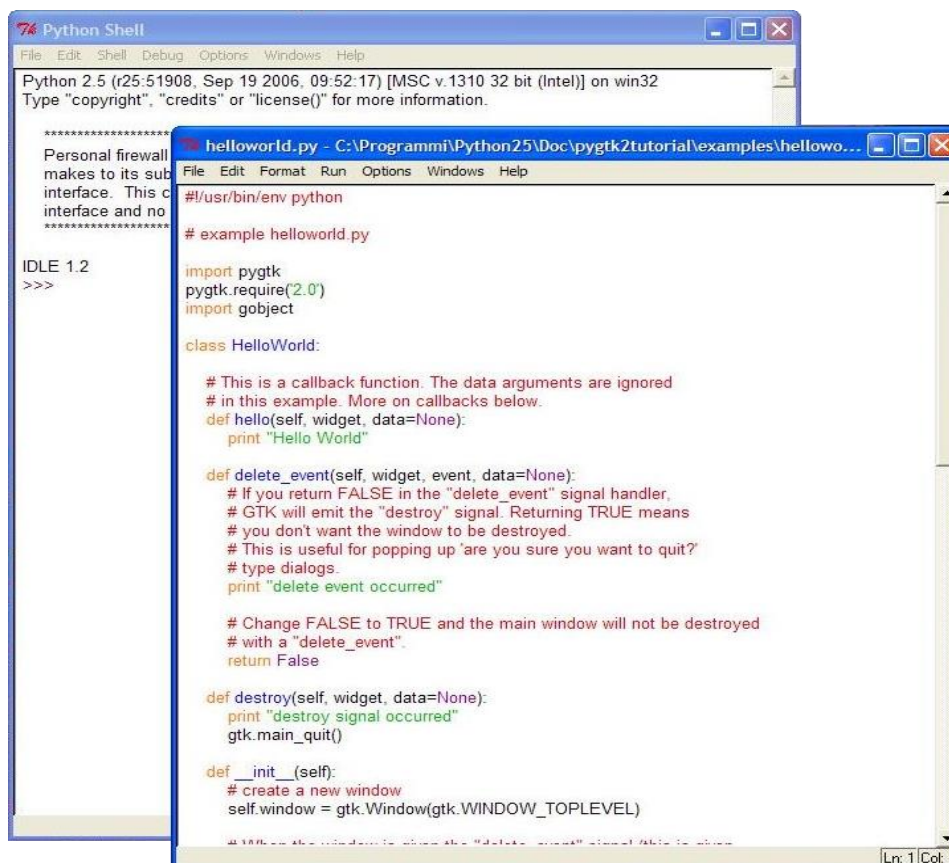


Clase 3

Programar en Python

Las bases de la sintaxis de Python



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall
makes to its sub
interface. This c
interface and no
*****

IDLE 1.2
>>>

helloworld.py - C:\Programmi\Python25\Doc\pygtk2tutorial\examples\hellowo...
File Edit Format Run Options Windows Help

#!/usr/bin/env python

# example helloworld.py

import pygtk
pygtk.require(2.0)
import gobject

class HelloWorld:

    # This is a callback function. The data arguments are ignored
    # in this example. More on callbacks below.
    def hello(self, widget, data=None):
        print "Hello World"

    def delete_event(self, widget, event, data=None):
        # If you return FALSE in the "delete_event" signal handler,
        # GTK will emit the "destroy" signal. Returning TRUE means
        # you don't want the window to be destroyed.
        # This is useful for popping up 'are you sure you want to quit?'
        # type dialogs.
        print "delete event occurred"

        # Change FALSE to TRUE and the main window will not be destroyed
        # with a "delete_event".
        return False

    def destroy(self, widget, data=None):
        print "destroy signal occurred"
        gtk.main_quit()

    def __init__(self):
        # create a new window
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

Una de las principales ventajas del uso de Python es su sintaxis clara y simple. En esta clase recorreremos los fundamentos básicos de la sintaxis de Python y formalizaremos algunos conceptos que nos permitirán comprender el uso del intérprete.

Familiarizándonos con el intérprete

Con el intérprete hemos tenido contacto básico en las pruebas realizadas luego de la instalación de python en los diferentes sistemas, pero a partir de esta clase comenzaremos a utilizarlo constantemente.

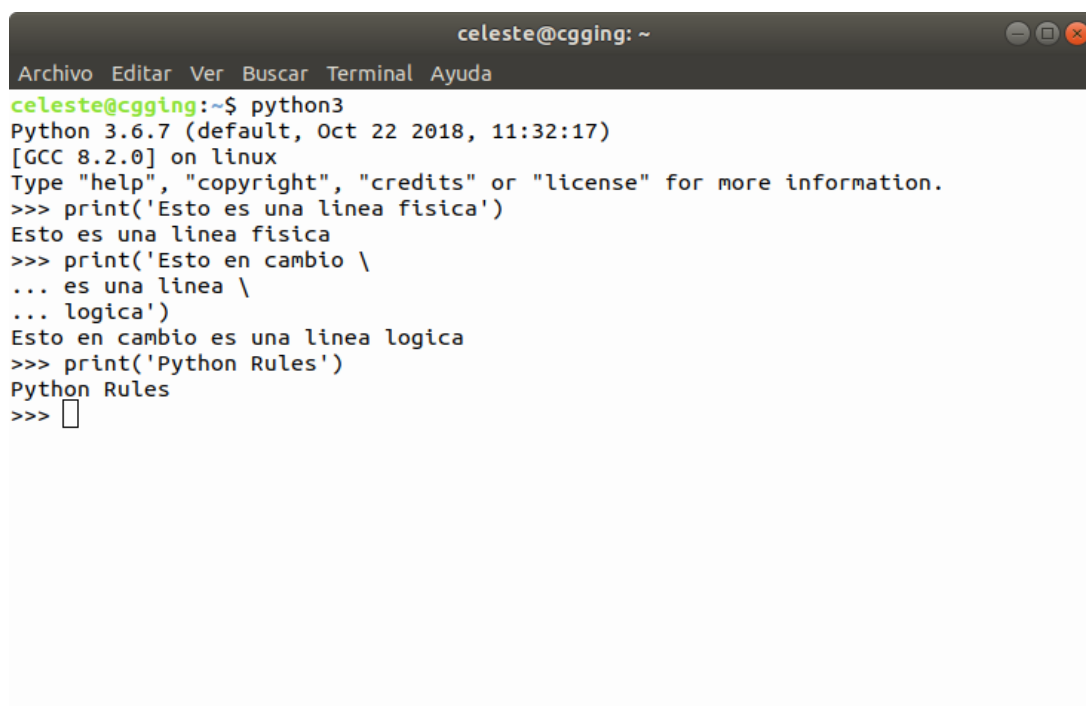
Una particularidad de la sintaxis de python consiste en el llamado “duck typing”, que es el estilo de tipificación de los datos que permite declarar implícitamente el tipo de dato de una variable en el momento de su asignación, permitiendo de esta manera que esta variable sea capaz de cambiar de tipo de dato a lo largo de su existencia.

Reglas generales.

Para comenzar a escribir código en Python, es preciso comprender algunas reglas generales de la sintaxis del lenguaje.

A continuación, se enumeran las principales reglas a tener en cuenta a la hora de realizar un código.

Líneas Físicas y Líneas Lógicas

A screenshot of a terminal window titled 'celeste@cgging: ~'. The window has a menu bar with 'Archivo', 'Editar', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'. The terminal shows the following text:

```
celeste@cgging:~$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Esto es una linea fisica')
Esto es una linea fisica
>>> print('Esto en cambio \
... es una linea \
... logica')
Esto en cambio es una linea logica
>>> print('Python Rules')
Python Rules
>>> 
```

Podemos entender por líneas físicas, a aquellas formadas por una secuencia de caracteres que terminan con un caracter de fin de línea (`\n` para sistemas Unix, o `\r\n` para Windows). Por ejemplo:

```
>>> print('Esto es una linea fisica')
Esto es una linea fisica
```

Es decir, en un archivo de texto plano, una línea física terminaría al presionar “enter”. Una línea lógica, en cambio, puede estar formada por varias líneas físicas. Hay dos formas de unir líneas físicas para formar una línea lógica, una forma implícita, la cual consiste en utilizar barra invertida ‘`\`’ colocada en la línea física que se quiere unir, justo antes del caracter de fin de línea; o la forma explícita, la cual consiste en encerrar las líneas físicas que se quieren unir utilizando los pares de caracteres: `()`, `[]`, `{}`.

Veamos un ejemplo:

Si una línea lógica se inicia con un paréntesis, se extenderá por tantas líneas físicas como sea necesario y solo terminará con el caracter de cierre. Esto también se cumple para los corchetes [] y las llaves {}.

La unión implícita es la forma recomendada y generalmente empleada por la mayoría de los programadores con experiencia en Python para poder visualizar el código de manera más cómoda.

Veamos un ejemplo de línea lógica declarada en forma implícita:

```
>>> print(  
... 'Hello',  
... 'World!'  
... )  
Hello World!
```

Construcción de las sentencias

Habiendo comprendido la diferencia entre líneas físicas y lógicas, solo cabe aclarar que en Python cuando hablemos de líneas, siempre nos estaremos refiriendo a líneas lógicas. entonces podemos comenzar a hablar sobre Sentencias

Una sentencia es una instrucción que el intérprete de Python puede ejecutar.

Las sentencias se pueden construir a partir de una línea física o de una línea lógica, dependiendo de la complejidad y composición de la instrucción que se desea ejecutar.

Sentencias Simples y Compuestas

Las sentencias de Python se componen de diferente número de líneas lógicas, sabiendo esto podemos realizar una clasificación sencilla de las sentencias entre simples y compuestas:

Las sentencias simples, son aquellas que deben completarse en una única línea lógica, como, por ejemplo:

```
>>> from sys import platform
```

Las sentencias compuestas, en cambio, son aquellas que deben comenzar con una condición de sentencia compuesta y deben contener sentencias simples y/o compuestas indentadas, a las cuales se les suele llamar cuerpo o bloque. La condición inicial o encabezado de una sentencia compuesta inicia siempre con una palabra reservada (*keyword*) y termina con el caracter de dos puntos.

Por ejemplo:

```
>>> if a > b:  
...     print(a, 'is greater than', b)
```

A diferencia de otros lenguajes, Python no tiene declaraciones u otros elementos sintácticos de alto nivel, solo sentencias, que generalmente ocupan una o varias líneas físicas en el editor que se utilice.

El fin de una línea física, generalmente determina el fin de la mayoría de las sentencias. Como se mencionó con anterioridad, las líneas físicas terminan con la secuencia de fin de línea `\n` en sistemas Unix o `\r\n` en Windows:

```
>>> var = 'Welcome to Python Scouts!' # Línea física que termina con la secuencia de fin de  
línea \n  
>>>
```

Otra forma de terminar una sentencia, y esto será familiar para aquellos que hayan tenido contacto con algún lenguaje de programación previamente, es emplear el uso del carácter punto y coma ‘;’ para terminar las sentencias:

```
>>> print(var); # Sentencia que termina con ;  
Welcome to Python Scouts!
```

Otro uso que se le puede dar a ‘;’ y que es el más difundido entre los programadores Python es incluir varias sentencias simples en una misma línea física:

```
>>> var1 = 0; var2 = 1 # Empleo del ; para separar dos sentencias en una misma línea física  
>>> var1  
0  
>>> var2  
1
```

Pero, deteniéndonos un segundo en este último uso, no es muy difícil entender que el empleo de ‘;’ para separar varias sentencias en una misma línea física atenta contra la legibilidad del código desarrollado en Python, haciendo engorrosa su interpretación por cualquier otro programador, o por la propia persona que escribió dicho código luego de un tiempo de realizado. Por este motivo el empleo de ‘;’ de esta manera se considera una mala práctica, y se recomienda evitarlo.

Buenas prácticas: un reglamento tácito entre programadores.

Cuando empezamos a trabajar con cualquier lenguaje de programación, tenemos que aprender las reglas básicas del lenguaje, si no respetamos la sintaxis, es de esperar que el intérprete o el compilador del lenguaje que estemos utilizando nos dé un error y no podamos ejecutar el código en cuestión.

Sin embargo, en la programación en general se fueron estipulando con los años reglas que se consideran *buenas prácticas* y que ayudan a la legibilidad del código de la misma forma que al trabajo colaborativo.

Generalmente el código funcionará igual si no respetamos estas buenas prácticas, pero a lo largo de este libro iremos viendo que su uso nos simplificará significativamente la tarea de programación.

Algunos ejemplos de buenas prácticas que podemos mencionar son:

Elegir nombres significativos para las variables.

Evitar la incorporación de más de una instrucción por línea.

Escribir los códigos de la manera más sencilla posible.

Otra buena práctica que cabe destacar en este punto es la de hacer uso del estilo de codificación estándar, ya que nos permite poder mostrar y consultar al respecto de nuestro código en distintas comunidades web de ser necesario.

A lo largo de este libro iremos explayándonos sobre el uso de buenas prácticas.

Indentación

El lenguaje Python, a diferencia de otros lenguajes, no emplea llaves ‘{}’ o estructuras begin...end para definir bloques de código. Para esto, el lenguaje se basa en el uso de lo que se conoce como indentación.

La indentación consiste en la inclusión de espacios o caracteres de tabulación al inicio de las líneas lógicas.

Los distintos niveles de indentación corresponden a los distintos bloques de programa.

La indentación del código tiene su origen en la necesidad de hacer que el código sea más legible y comprensible. Esta es la razón fundamental por la cual Python la incluye directamente como parte de su sintaxis, esencialmente para mejorar la legibilidad, comprensión y sencillez del código. Este es uno de los rasgos identificativos y más valorados del lenguaje.

Las sentencias pueden ser agrupadas dentro de una cláusula o cabecera (header) de sentencia compuesta mediante la indentación.

De este modo, Python usa la indentación de las líneas lógicas, para determinar la agrupación de sentencias y su pertenencia a determinado bloque o cuerpo de sentencia compuesta. Por ejemplo:

def printer():

Los espacios al inicio de estas líneas forman la indentación

print('Hello World!')

print('Welcome to Python Scouts!')

Las sentencias anteriores forman el bloque o cuerpo de la función printer()

printer() # El llamado a printer() queda fuera del bloque, pues ya no hay indentación

La indentación puede definirse con caracteres de espacio (se recomienda el empleo de 4 espacios, que es considerado el estilo óptimo de Python) o de tabulación. Es recomendable no mezclar ambos tipos de caracteres en un mismo fragmento de código. De hecho, dependiendo de cómo esté configurado el editor de texto algunos intérpretes pueden devolver error en este caso. La indentación debe ser la misma (igual cantidad de espacios), al menos para las líneas que componen un mismo bloque de código y la primera sentencia de un archivo de código no debe tener indentación.

Como se puede observar, la indentación es un componente fundamental en la sintaxis de Python.

Si bien en otros lenguajes de programación la indentación es considerada una buena práctica, en Python es una característica fundamental del lenguaje que de utilizarse en forma errónea producirá que el código no pueda ejecutarse, por lo que particularmente en Python no se considera puntualmente como una buena práctica sino como una característica obligatoria de la sintaxis.

Comentarios

```
celeste@cgging: ~
Archivo Editar Ver Buscar Terminal Ayuda
celeste@cgging:~$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Esto es una línea física')
Esto es una línea física
>>> print('Esto en cambio \
... es una línea \
... lógica')
Esto en cambio es una línea lógica
>>> print('Python Rules')
Python Rules
>>> #podemos poner tantos comentarios como queramos
... print('texto')#tanto antes como al final de una línea
texto
>>> print('# esto no es un comentario')
# esto no es un comentario
>>> 
```

Los comentarios consisten en aclaraciones o indicaciones para el desarrollador o usuario a lo largo del código que resulten de utilidad. En general son secuencias de caracteres que comienzan con el caracter de numeral # y continúan hasta el fin de la línea física.

Las líneas físicas que comienzan con # son ignoradas completamente por el intérprete de Python. En realidad, estas líneas son muy importantes y están dirigidas a los programadores/mantenedores/clientes del código. En muchas ocasiones seremos nosotros los consumidores de estos comentarios, y nos facilitarán considerablemente la tarea de comprender, reutilizar o mantener un proyecto, sobre todo si estamos hablando de proyectos grandes o de largo plazo.

Los comentarios son útiles para indicar lo que se está realizando en el código y sobre todo por qué se lo hace, no obstante, una buena práctica también consiste en saber cuándo incluir un comentario, y no ‘ensuciar’ el código con comentarios superfluos o redundantes que no aporten información importante para el proyecto. Por ejemplo:

```
>>> # Esto es un comentario que comienza con # y es ignorado por Python
>>>
>>> # Le asigno el valor 0 a count <= Esto es un comentario innecesario
>>> count = 0
>>>
>>> # Inicializo count a 0 para contar las veces que... <= Mejor
>>> count = 0
```

Lo que vimos hasta ahora en el ejemplo, son comentarios que se anotan en una línea separada del código. A veces resulta más útil realizar un comentario en línea. El comentario en línea se realiza a continuación de la sentencia o línea que se quiere explicar.

Veamos un ejemplo:

```
>>> if temperature <= 273: # Validar la temperatura <= Comentario en línea
>>>     raise ValueError('Wrong temperature value')
```

Uniando las líneas físicas en forma implícita, como se explicó con anterioridad, se pueden incluir comentarios en línea de la siguiente manera:

```
month_names = ['January', 'February', 'March', # Primer trimestre
               'April', 'May', 'June', # Segundo trimestre
               'July', 'August', 'September', # Tercer trimestre
               'October', 'November', 'December'] # Cuarto trimestre
```

Tokens del Lenguaje

Los *tokens* de Python, son componentes fundamentales del lenguaje que forman las líneas lógicas. Ya se ha trabajado con tokens a lo largo de este capítulo, pero no los habíamos

mencionado formalmente aún. A modo de ejemplo y para comprender mejor qué es un token, podemos mencionar:

NEWLINE: determina el fin de una línea lógica y el comienzo de otra.

INDENT: indentación de las sentencias dentro de una sentencia compuesta.

DEDENT: fin de indentación que determina el fin de una sentencia compuesta.

Estos tres tokens que se acaban de mencionar fueron utilizados a lo largo de todo el capítulo y se explicó tanto su uso como su importancia. Pero no son los únicos tokens del lenguaje. Algunos elementos del lenguaje facilitan la creación de distintas estructuras. Entre ellos podemos mencionar el uso de *identificadores*, otro token del lenguaje.

Identifiers, o identificadores son aquellos nombres que identifican a variables, funciones, clases, métodos, constantes, módulos, paquetes, etc. Dichos tokens se comienzan con letras que pueden ser mayúsculas o minúsculas o con guion bajo y luego pueden contener dígitos, letras o más guiones bajos. Es importante tener en cuenta que Python es un lenguaje Case Sensitive, lo que significa que las letras mayúsculas son distintas de las minúsculas. Por ejemplo:

```
>>>a=5 #se declara la variable 'a' y se le asigna el valor 5
>>>A=3 # se declara otra variable independiente de la anterior 'A' y se le asigna el valor 3
>>>a+A
8
```

Otro token que podemos identificar y que hemos mencionado con anterioridad se refiere a las Palabras Reservadas (keywords): palabras con significado especial para el lenguaje, que no pueden ser empleadas como identificadores. Algunas palabras clave son sentencias simples (Ej. break, continue), otras, condiciones de sentencias compuesta (Ej. def, class, for, while), mientras que otras son operadores (Ej. and, or, is, in). Las palabras reservadas de Python se pueden consultar tecleando en el prompt del intérprete, las sentencias siguientes:

```
>>> import keyword
```

```
>>> keyword.kwlist
```

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



```
celeste@cgging: ~
Archivo Editar Ver Buscar Terminal Ayuda
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

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

help>
```

También encontramos a los Literales o literals, valores numéricos o de cadena de caracteres que aparecen directamente escritos en el código. Por ejemplo:

```
>>> 'Hello World!' # Literal de string
```

```
'Hello World!'
```

```
r>>> 1452.25 # Literal de float
```

```
1452.25
```

```
>>> 15 # Literal de int
```

```
15
```

```
>>> 1_000_000 # Literal de int con guión bajo de agrupación (versión 3.6)
1000000
```

Otro elemento importante y que tal vez no nos imaginaríamos que podían clasificarse como token, son los Operadores u operators. Estos son caracteres empleados para denotar operaciones diversas tales como: aritméticas, lógicas, de asignación, etc.

Los operadores actuales del lenguaje son:

`+` `-` `*` `**` `/` `//` `%` `@`

`<<` `>>` `&` `|` `^` `~`

`<` `>` `<=` `>=` `==` `!=`

Y finalmente, pero no por eso menos importante, encontramos a los *Delimitadores o delimiters*: caracteres empleados para delimitar literales, líneas lógicas, entre otras. Python incluye los siguientes delimitadores:

`(` `)` `[` `]` `{` `}`

`,` `:` `.` `;` `@` `=` `->`

`+=` `-=` `*=` `/=` `//=` `%=` `@=`

`&=` `|=` `^=` `>>=` `<<=` `**=`

```
celeste@cgging: ~
Archivo  Editar  Ver  Buscar  Terminal  Ayuda

Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

(END)
```

Lo que aprendimos en esta clase

En esta clase vimos las bases del lenguaje Python y las plasmamos en diferentes ejemplos mediante fragmentos de códigos, pudiendo de esta manera comprender la diferencia entre línea física y lógica, cómo construir sentencias, que es un token y profundizamos sobre cada uno de estos.

Además, profundizamos los comandos y el uso del intérprete y comenzamos a entender el concepto de buenas prácticas en la programación, concepto sobre el que nos seguiremos explicando y aplicando en clases siguientes.