Introducción a Python 3.6

Antonio Gámiz Delgado

1 de noviembre de 2017

1. Instalación
2. Primeros pasos
   - Intérprete
   - Programar en un fichero a parte
   - UTF-8
   - Hola, mundo
3. Variables
   - Definición de variables
   - Reglas de nomenclatura
   - Tipos de variables
4. Estructuras de control básicas.
   - Operadores lógicos
   - Estructura If
   - Estructura While
   - Estructura For
5. Funciones y programación dirigida a objetos

Instalación

En Linux, Pyhton viene normalmente instalado en la distribución base que uséis(Ubuntu, Fedora, Guadalinex, etc, ya que es el lenguaje con el que se escriben muchas de las aplicaciones base que llevan.
En este tutorial usaremos *Python*3.6

Para ver la versión de Python que usáis:

python –version
sudo apt-get install python3.6

La instalación de *Python* sólo nos proporciona un intérprete que permite ejecutar los programas escritos en él, por lo que necesitamos un editor:

La instalación de *Python* sólo nos proporciona un intérprete que permite ejecutar los programas escritos en él, por lo que necesitamos un editor:

Creamos un fichero: miprograma.py

Para ejecutar un programa en *python* directamente desde la línea de comandos y sin necesidad de llamar al intérprete añadimos lo siguiente:

```
1 #!/usr/bin/python3.6
```

Creamos un fichero: miprograma.py

Para ejecutar un programa en *python* directamente desde la línea de comandos y sin necesidad de llamar al intérprete añadimos lo siguiente:

```
1 #!/usr/bin/python3.6
```

Además de crearlo, debemos darle permisos de ejecución:
chmod +x miprograma.py

UTF-8

Desde *python*3, las cadenas por defecto son $Unicode(UTF-8)$, pero si usáis una versión anterior a la 3, tendréis que añadir la siguiente línea a vuestro archivo.py:

```
1 #!/usr/bin/python
2 #-*- coding: utf-8 -*-
```

Nuestro primer programa

Con la función *print* crearemos nuestro primer programa:

```
1 #!/usr/bin/python
2 #-*- coding: utf-8 -*-
3
4 print("Hola, mundo")
```

Nuestro primer programa

Con la función *print* crearemos nuestro primer programa:

```
1 #!/usr/bin/python
2 #-*- coding: utf-8 -*-
3
4 print("Hola, mundo")
```

Algún parámetro de *print*

end="delimitador"

```
1 print("Hola, mundo",end="")
```

¿Cómo se definen?

En *python* las variables se definen de la siguiente forma:
nombre_de_la_variable=valor

¿Cómo se definen?

En *python* las variables se definen de la siguiente forma:
nombre_de_la_variable=valor

Algunos ejemplos:

```
1 t=2.5
2 nombre="Esto es una frase"
3 caracter='a'
4 meses=['Enero','Febrero']
5 random=[1,'patata',30234,'a']
6 masrandom=[1,'patata',30234,'a',['Enero','Febrero']]
7 a = b = 69
8 x, y, z = "hola", 10, ["teclado", "patatas"]
```

Reglas de nomenclatura

Letras permitidas

Las letras permitidas para nombrar variables son las del 'alfabeto inglés', es decir, ni la 'ñ' ni la 'ç' están permitidas.

Python distingue entre minúsculas y mayúsculas.

Reglas de nomenclatura

Letras permitidas

Las letras permitidas para nombrar variables son las del 'alfabeto inglés', es decir, ni la 'ñ' ni la 'ç' están permitidas.

Python distingue entre minúsculas y mayúsculas.

El nombre de una variable puede empezar por una letra o por '_', pero no por un número. Y puede seguir con más letras, números o '_'.

Las palabras reservadas del lenguaje están prohibidas(if, else, while, etc).

Algunos tipos son:

```
1 x=2.5
2 nombre="Esto es una frase"
3 caracter='a'
4 meses=['Enero','Febrero']
5 random=[1,'patata',30234,'a']
6 diccionario={"a":"a", "b":"c"}
```

Algunos tipos son:

```
1 x=2.5
2 nombre="Esto es una frase"
3 caracter='a'
4 meses=['Enero','Febrero']
5 random=[1,'patata',30234,'a']
6 diccionario={"a":"a", "b":"c"}
```

type(*var*):Devuelve el tipo de *var*.

```
1 x=1
2 y='a'
3 z=[a,b,c]
4
5 print(type(x)," ",type(y)," ",type(z))
```

Conversión de variables

Python es un lenguaje fuertemente tipado, es decir, no se puede tratar a una variable como si fuera de un tipo distinto al que tiene.

Conversión de variables

Python es un lenguaje fuertemente tipado, es decir, no se puede tratar a una variable como si fuera de un tipo distinto al que tiene.

Solución: consersiones

type(variable)

```
1 x=1
2 y='a'
3
4 str(x)
5 int(a)
6
7 print(type(x)," ",type(y))
```

Introduciendo variables por teclado

Función input

En Python3.x: input() En Python2.x: raw_input() input()

```
1
2 #!/usr/bin/python3.6
3
4 x1=int(input("numero 1: "))
5 x2=int(input("numero 2: "))
6
7 print("La suma de los dos es: ", x1+x2)
```

Operadores aritméticos

```
1 #!/usr/bin/python3.6
2
3 x + y #suma
4 x - y #resta
5 x * y #multiplicacion
6 x / y #division
7 x // y  #division entera
8 x %y #modulo
9 x ** y  #exponente
```

Listas(I).

```
1 #!/usr/bin/python3.6
2 lista = [1,2,3,4,5]
3 lista2 = [1,2,3,[4,5,6]]
4
5 print(lista[2])
6 print(lista[-1])
7 print(lista[14])
8 print(lista2[3][0])
```

Listas(I).

```
1 #!/usr/bin/python3.6
2 lista = [1,2,3,4,5]
3 lista2 = [1,2,3,[4,5,6]]
4
5 print(lista[2])
6 print(lista[-1])
7 print(lista[14])
8 print(lista2[3][0])
```

Error

Traceback (most recent call last): File "nombres.py", line 26, in ¡module¿ print(nombres[-5]) IndexError: list index out of range

Listas (II).

```
1 #!/usr/bin/python3.6
2 lista = [1,2,3,4,5]
3 lista2=list(lista) #Importante!!
4 lista[1]=3  #Modificamos lista.
5 lista.append("otroElemento") #Aniadimos un nuevo
      elemento a 'lista'.
6 lista.insert(1, "otroElemnto2") #Aniadimos un nuevo
      elemento en la posicion 1.
7 lista.pop()  #Quita el ultimo elemento de 'lista' y lo
       retorna.
```

Listas (II).

```
1 #!/usr/bin/python3.6
2 lista = [1,2,3,4,5]
3 lista2=list(lista) #Importante!!
4 lista[1]=3  #Modificamos lista.
5 lista.append("otroElemento") #Aniadimos un nuevo
      elemento a 'lista'.
6 lista.insert(1, "otroElemnto2") #Aniadimos un nuevo
      elemento en la posicion 1.
7 lista.pop()   #Quita el ultimo elemento de 'lista' y lo
       retorna.
```

Pila

Intuitivamente, se puede ver que con append() y pop() se puede hacer una 'pila' fácilmente.

Diccionarios.

```
1  #!/usr/bin/python3.6
2
3  mydict = {"altura" : "media", "habilidad" : "alta", "
       salario" : 999}
4
5  #Python2.x
6  if mydict.has_key('altura'):
7  print 'Llave encontrada'
8
9  #Python3.x
10 if 'altura' in mydict:
11 print("Llave encontrada")
```

Operadores lógicos

```
1  x < y
2  x > y
3  x == y
4  x >= y
5  x <= y
6  x != y
7  x in lista
8  x not in lista
9  x is y #id(x)==id(y)
10 x is not y
11 exp1 and exp2
12 exp1 or exp2
13 exp1 not exp2
```

Estructura if

```
#!/usr/bin/python3.6

if expresion_a_evaluar:
  instruccion1
  ...
else:
  instruccion2
  ...

if expresion_a_evaluar_1:
  ejecutar_si_cierto_1
elif expresion_a_evaluar_2:
  ejecutar_si_cierto_2
elif expresion_a_evaluar_3:
  ejecutar_si_cierto_3
else:
  ejecutar_si_ninguna
```

Estructura While

```
1 #!/usr/bin/python3.6
2
3 while condicion:
4   instruccion
5
6 while condicion:
7   instruccion
8 else:
9   instruccion_fuera_del_bucle
```

Estructura For

For

El bucle 'for' sirve para recorrer secuencialmente los elementos de una lista.

```
1  #!/usr/bin/python3.6
2  for variable in lista:
3    instrucciones
4  else:
5    instrucciones_si_llega_al_final_del_bucle
6
7  #Ejemplo:
8  Huerto = ["zanahoria", "col", "lechuga", "col"]
9  for Planta in Huerto:
10   if Planta != "col":
11     print(Planta)
12 else:
```

Sentencias 'break' y 'continue'

Break

'break' sale completamente del bucle sin ejecutar ninguna instrucción más(ni 'else's)

Continue

'continue' es más suave que 'break', se salta la iteración en la que se activa, pero luego sigue ejecutando el bucle.

```
1 #!/usr/bin/python3.6
2
3 x= 0
4 while x < 10:
5   if x == 5:
6     break (o 'continue')
7   print(x)
```

Errores varios

Cosas como la división por cero o el tratamiento de tipos de datos incompatibles (sumar cadenas, por ejemplo) provocarán errores en tiempo de ejecución que abortaran el programa.

Errores varios

Cosas como la división por cero o el tratamiento de tipos de datos incompatibles (sumar cadenas, por ejemplo) provocarán errores en tiempo de ejecución que abortaran el programa.

¿Cómo solucionarlo? 'try'

```
1  try:
2    instruccionesaejecutar
3  except:
4    aqui_entra_por_cualquier_error
5  try:
6    instrucciones_a_ejecutar
7  except TipoDeError1:
8    ...
9  except TipoDeErrorN:
10   ...
11 else:
```

Gestión de errores (ejemplo)

```
try:
  resultado = dividendo/divisor
except ZeroDivisionError:
  if divisor == 0:
  print("No puedes dividir por cero.")
except ValueError:
  if divisor == 0:
  print("0 no es valido")
else:
  print("La divison resulta: ", resultado)
```

Funciones

```
1 def nombre_funcion(arg1, arg2, ..., argN):
2   cuerpo
3   return algo (si hubiera que devolver algo)
```

return

Aunque la función no devuelva ningún valor, siempre devolverá 'None'(como el 'null' de Java).

```
1 def cuadrado(x):
2   return x**2
```

Funciones (II): Argumentos por defecto.

```
1 #!/usr/bin/python3.6
2 def mostrar(mensaje="argumento por defecto"):
3     print(mensaje)
4
5 mostrar()
6 mostrar("otro mensaje")
```

Funciones (II): Argumentos por defecto.

```
1 #!/usr/bin/python3.6
2 def mostrar(mensaje="argumento por defecto"):
3     print(mensaje)
4
5 mostrar()
6 mostrar("otro mensaje")
```

¡Importante!

Los argumentos opcionales sólo se pueden ir al final ya que se acomodan de izquierda a derecha.

Programación dirigida a objetos

¿Qué es una clase?

Una clase es un tipo de dato definido por el usuario, que podemos ejemplificar (instanciar) para obtener instancias, es decir, objetos de ese tipo.

Programación dirigida a objetos

¿Qué es una clase?

Una clase es un tipo de dato definido por el usuario, que podemos ejemplificar (instanciar) para obtener instancias, es decir, objetos de ese tipo.

Características

1. Podemos llamar a un objeto de clase como si fuera una función.
2. Una clase tiene atributos nominales a los que podemos hacer referencia.
3. Los atributos de la clase vinvulados a funciones se conocen como 'métodos de la clase'.
4. Una clase puede heredar de otras clases

Creando nuestras primeras clases.

```
1  #!/usr/bin/python3.6
2
3  class nombre_clase():
4    atributos
5    metodos
6
7  class ejemplo():
8    x=25
9    def metodo_mostrar(self):
10     print("El valor de x es: ",x)
```

Constructor de la clase (Ej.1)

```
1  #!/usr/bin/python3.6
2  class Acuario():
3    #El metodo __init__ se ejecuta automaticamente al
       llamar a la clase
4    def __init__(self, litros, tipo, num):
5      self.capacidad = litros
6      self.tipo = tipo
7      self.total_peces = num
8    def meter_pez(self):
9      self.total_peces = self.total_peces + 1
10   def sacar_pez(self):
11     if ( self.total_peces > 0):
12       self.total_peces = self.total_peces + 1
```

Constructor de la clase (Ej.1)

```
1
2  Mipecera= Acuario(100,"tropical",5)
3
4  print ("La pecera tiene", Mipecera.capacidad, "litros")
5  print ("Hay", Mipecera.total_peces, "peces")
6  print ("Aniadimos un pez")
7
8  Mipecera.meter_pez()
9
10 print ("Ahora hay", Mipecera.total_peces, "peces")
```

Constructor de la clase (Ej.2)

```
1  #!/usr/bin/pyhton3
2
3  class Complejo:
4    def __init__(self, parteReal, parteImaginaria):
5      """Constructor de la clase Complejo"""
6      self.r = parteReal
7      self.i = parteImaginaria
8    def getReal(self):
9      return self.r
10   def getImag(self):
11     return self.i
12
13 num = Complejo(3.0, -4.5)
14 print ("Parte Real ", num.getReal())
15 print ("Parte Imaginaria ", num.getImag())
```

Herencia, (PyRencia)

¿Qué es la herencia?

En POO, la herencia es, después de la agregación o composición, el mecanismo más utilizado para alcanzar algunos de los objetivos más preciados en el desarrollo de software como lo son la reutilización y la extensibilidad. A través de ella los diseñadores pueden crear nuevas clases partiendo de una clase o de una jerarquía de clases preexistente (ya comprobadas y verificadas) evitando con ello el rediseño, la modificación y verificación de la parte ya implementada.

Ejemplo de herencia

```
1  #!/usr/bin/pyhton3
2  class Mamifero():
3    def __init__(self):
4      descripcion= "Los mamiferos son interesantes"
5    def tiene_pelo(self):
6      return "si"
7    # Al poner Mamifero en el parentesis, le decimos que
       herede de esa calse
8  class Gato(Mamifero):
9    def __init__(self):
10     descrpcion= "Los gatos molan"
11   def maulla(self):
12     return "si"
13 # Mascota es una instancia de la clase Gato
14 Mascota = Gato()
15 # Estamos llamando a un metodo de la clase mamifero
16 print (Mascota.tiene_pelo())
```

Módulos (I)

¿Qué es un módulo?

Un módulo es una parte de algo. Algo es modular si es posible separarlo en partes o piezas, como los LEGO.

Módulos (I)

¿Qué es un módulo?

Un módulo es una parte de algo. Algo es modular si es posible separarlo en partes o piezas, como los LEGO.

¿Cómo son los módulos en python?

En Python, un módulo es una pequeña pieza de un gran programa. Sencillamente un módulo es un fichero dentro de tu disco duro. En caso de Python es un fichero con extension *.py*.

Módulos (I)

¿Qué es un módulo?

Un módulo es una parte de algo. Algo es modular si es posible separarlo en partes o piezas, como los LEGO.

¿Cómo son los módulos en python?

En Python, un módulo es una pequeña pieza de un gran programa. Sencillamente un módulo es un fichero dentro de tu disco duro. En caso de Python es un fichero con extension *.py*.

¿Cómo usarlos?

Sentencia *import*, *import as*, *from import*

Módulos (II)

Módulos ya hechos de Python

Python tiene instalada por defecto una gran cantidad de módulos que amplían su uso. Además, por supuesto, es posible instalar nuevos módulos.

¿Cómo saber que módulos tenemos instalados?

1. Iniciamos una sesión interactiva (*python* o *python3*)
2. Escribimos *help*() *modules*

Módulos (III)

Información de un módulo específico.

1. Iniciamos una sesión interactica (*python3*)
2. import módulo
3. help(módulo)
4. help(módulo.funcion)

Ver las funciones de un módulo

1. Iniciamos una sesión interactica (*python3*)
2. import módulo
3. dir(módulo)