Práctica 1 - Introducción a Python

Protocolos para la Transmisión de Audio y Vídeo en Internet

Gregorio Robles

{grex,jgb}@gsyc.urjc.es
GSyC, Universidad Rey Juan Carlos

30 de septiembre de 2020

Python

Transparencias principales

Transparencias principales

(las que veremos en clase)

Hola Mundo

- Desde la shell, accede al intérprete de Python **3**:

```
$ python3
>>>
```

- Y ya podemos introducir instrucciones en Python:

```
>>> print("hola mundo")
hola mundo
```

- A partir de ahora obviaremos generalmente los >>> del intérprete.

Más ejemplos

- Podemos usar Python como calculadora
- Verás que Python es sensible mayúsculas
- Los comentarios se indican con #
- En Python hay diferentes tipos de datos

```
print("hola mundo")    # esto es un comentario
euros = 415
pesetas = euros * 166.386
print(str(euros) + " euros son "+ str(pesetas) + " pesetas")
```

Sangrado y separadores de sentencias

- ¡En Python NO hay llaves ni `begin-end` para encerrar bloques de código!
- Un mayor nivel de sangrado indica que comienza un bloque, y un menor nivel indica que termina un bloque.
- Ejemplo:

```
# Ejemplo de dos funciones en Python
def a_centigrado(faren):
    """Convierte grados farenheit en grados centígrados"""
    return (faren - 32) * (5.0/9)


def a_farenheit(cels):
    """Convierte grados centígrados en grados farenheit"""
    return (cels * 1.8) + 32
```

Condicional

Sentencia `if`:

```
entero = 3
if entero:
     print('verdadero')
else:
     print('falso')
```

Nótese como el caracter `:` introduce cada bloque de sentencias. Si hay `:`, entonces la siguiente línea estará indentada.

Cadenas

- No existe tipo `char`
- Comilla simple o doble
  `print("hola")` o `print('hola')`
  `print('me dijo "hola"')`
  más legible que `print('me dijo \'hola\'')`
- Puede haber caracteres especiales
  `print("hola\nque tal")`
- El operador + concatena cadenas, y el * las repite un número entero de veces

Listas

- Tipo de datos predefinido en Python, va mucho más allá de los arrays
- Es un conjunto **indexado** de elementos, no necesariamente homogéneos
- Sintaxis: Identificador de lista, mas índice entre corchetes
- Cada elemento se separa del anterior por un caracter ,

```
hijos = ['Gregor','Cayetana']
hijos.append('Manuel')
print(hijos)
print(hijos[2])
print(len(hijos))

cosas = ['uno', 2, 3.0]
```

Más sobre listas

- El primer elemento tiene índice 0.
- Un índice negativo accede a los elementos empezando por el final de la lista. El último elemento tiene índice -1.
- Pueden referirse **rodajas** (*slices*) de listas escribiendo dos índices entre el caracter :
- La rodaja va desde el **primero, incluido**, al **último, excluido**.
- Si no aparece el primero, se entiende que empieza en el primer elemento (0)
- Si no aparece el segundo, se entiende que termina en el último elemento (incluido).

Ejemplos de listas

```
lista = [0, 1, 2, 3, 4]
print(lista)      # [0, 1, 2, 3, 4]
print(lista[1])   # 1
print(lista[0:2]) # [0, 1]
print(lista[3:])  # [3, 4]
print(lista[-1])  # 4
print(lista[:-1]) # [0, 1, 2, 3]
print(lista[:-2)] # [0, 1, 2]
```

¡La misma sintaxis se aplica a las cadenas!

```
cadena = "estudiante"
print(cadena[-1])
```

Bucles

Sentencia `for`:

```
>>> amigos = ['ana', 'jacinto', 'guillermo']
>>> for invitado in amigos:
...     print(invitado, len(invitado))
...
ana 3
jacinto 7
guillermo 9
```

Diccionarios

- Es un conjunto **desordenado** de elementos
- Cada elemento del diccionario es un par clave-valor.
- Se pueden obtener valores a partir de la clave, pero no al revés.
- Longitud variable
- Elementos heterogéneos
- Hace las veces de los *registros* en otros lenguajes
- Atención: Se declaran con `{}`, se refieren con `[]`

Más sobre diccionarios

```
paises = {'de': 'Alemania', 'fr': 'Francia', 'es': 'España'}
print(paises); print(paises["fr"])

extensiones = {}
extensiones['py'] = 'python'
extensiones['txt'] = 'texto plano'
extensiones['doc'] = 'Word'

for pais in paises: # iteramos por el diccionario
   print(pais, paises[pais])

del paises['fr']   # borra esta llave (y su valor)
print(len(paises)) # devuelve el número de elementos en el diccionario
paises.clear()  # vacía el diccionario
```

Sobre los diccionarios

- Asignar valor a una clave existente reemplaza el antiguo
- Una clave de tipo cadena es sensible a mayúsculas/minúsculas
- Pueden añadirse entradas nuevas al diccionario
- Los diccionarios se mantienen desordenados
- Los valores de un diccionario pueden ser de cualquier tipo
- Las claves pueden ser enteros, cadenas y algún otro tipo
- Pueden borrarse un elemento del diccionario con `del`
- Pueden borrarse todos los elementos del diccionario con `clear()`

Características

Parémonos a ver las características de Python. Python es:

- de alto nivel
- interpretado (no compilado)
- orientado a objetos (todo son objetos)
- dinámicamente tipado (frente a estáticamente tipado)
- fuertemente tipado (frente a débilmente tipado)
- sensible a mayúsculas/minúsculas

Utilizando un editor o un IDE (I)

- Usar el intérprete de Python para programar es tedioso
- Es mejor utilizar cualquier editor de texto (p.ej., gedit) o un IDE (como Eclipse)
- Lo que crearemos son ficheros de texto plano.
- Se puede añadir información en la parte superior del fichero para indicar a la shell que es un fichero en Python y con caracteres UTF-8 (y así añadir eñes y tildes, p.ej.).

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

print("¡Hola Mundo!")
```

Utilizando un editor o un IDE (y II)

- Si lo guardamos como hola.py, se ejecuta desde la línea de comandos como:

  ```
  $ python3 hola.py
  ```

- Podemos darle permisos de ejecución al fichero:

  ```
  $ chmod +x hola.py
  ```

- Y entonces, se ejecuta desde la línea de comandos como:

  ```
  $ ./hola.py
  ```

Ficheros

- `open(nombre_fichero, modo, encoding)` devuelve un objeto fichero
  modo:
  - `w`: Escritura. Destruye contenido anterior
  - `r`: Lectura. Modo por defecto
  - `r+`: Lectura y Escritura
  - `a`: Append

  codificación (opcional): Por ejemplo, encoding='utf-8'
- `write(cadena)` escribe la cadena en el fichero
- `read()` devuelve el contenido del fichero
- `readlines()` devuelve una lista con cada línea del fichero
- `readline()` devuelve la siguiente línea del fichero
- `close()` cierra el fichero

Ejemplos de uso de ficheros

```
#!/usr/bin/python3
fich=open('/tmp/prueba','w')
fich.write("lunes\n")
fich.close()

fich=open('/tmp/prueba','a')
fich.write("martes\n")
fich.close()

fich=open('/etc/hosts','r')
maquinas=fich.readlines()
fich.close()

for maquina in maquinas:
    print(maquina)
```

Un programa en Python

```
def sum(sumando1, sumando2):
    """Sums two integer/floats

    Returns integer/float."""

    return sumando1 + sumando2

if __name__ == "__main__":
    primero = int(raw_input("Please enter an integer/float: "))
    segundo = int(raw_input("Please enter another integer/float: "))
    print(sum(primero, segundo))
```

Ejecución:

```
$ python3 suma.py
```

El atributo __name__ de un módulo

Los módulos son objetos, con ciertos atributos predefinidos.
El atributo `__name__`:

- si el módulo es importado (con `import`), contiene el nombre del fichero, sin trayecto ni extensión
- si el módulo es un programa que se ejecuta sólo, contiene el valor `__main__`

Puede escribirse ejecución condicionada a cómo se use el módulo:

```
if __name__ == "__main__":
    ...
```

Importar módulos

- `import nombre-módulo`
  permite acceder a los símbolos del módulo con la sintaxis `nombre-módulo.X`
- `from nombre-módulo import a, b, c`
  incorpora los símbolos a, b, c al espacio de nombres, siendo accesibles directamente (sin cualificarlos con el nombre del módulo)
- `from nombre-módulo import *`
  incorpora los símbolos del módulo al espacio de nombres, siendo accesibles directamente (sin cualificarlos con el nombre del módulo).

Test Unitarios

- La automatización de tests tiene muchas ventajas
- Tantas, que hay gente que incluso dice que ¡primero deberíamos escribir los tests!
- En Python, se pueden escribir tests unitarios con ayuda del módulo `unittest`

Test Unitarios

```
import calc
import unittest

class TestAdd(unittest.TestCase):
    """
    Test the add function
    """

    def test_add_integers(self):
        """
        Test that the addition of two integers
        """
        result = calc.add(1, 2)
        self.assertEqual(result, 3)

if __name__ == '__main__':
    unittest.main()
```

Referencias

Si necesitas practicar más:

- Code Academy for Python
  https://www.codecademy.com/learn/learn-python
- Playground and cheatsheet for learning Python
  https://github.com/trekhleb/learn-python
- Awesome Python
  https://github.com/vinta/awesome-python

Consideraciones adicionales

Consideraciones Adicionales

(transparencias de referencia)

Ámbito de las variables

- Las variable declaradas fuera de una función son globales

```
#!/usr/bin/python3
numero = 5
def f(parametro):
    return parametro + numero
print(f(3))    # 8
```

- Las variable declaradas dentro de una función son locales

```
#!/usr/bin/python3
def f(parametro):
    numero = 5
    return parametro + numero
print(f(3))
print(numero)    # ERROR: numero es de ambito local
```

Más sobre ámbito de variables

- Dentro de una función se puede ver una variable global pero no modificar

```
#!/usr/bin/python3
numero = 5
def f(parametro):
    numero = numero-1    #ERROR: no se puede modificar variable global
    return paramentro + numero

print(f(3))
```

- A menos que se use la sentencia global

```
#!/usr/bin/python3
numero = 5
def f(parametro):
    global numero   #permite modificar una variable global
    numero = numero-1
    return parametro + numero
```

Más sobre ámbito de variables

- Un poco más complicado:

```
#!/usr/bin/python3
numero = 5
def f(parametro):
    numero = 4   # ahora numero es variable local
    return parametro + numero

print(f(3))   # 7
print(numero) # 5
```

Definición de variables

Python es

- fuertemente tipado (frente a débilmente tipado)
- sensible a mayúsculas/minúsculas

En Python la declaración de variables es implícita
(no hay declaración explícita)

- Las variables "nacen" cuando se les asigna un valor
- Las variables "desaparecen" cuando se sale de su ámbito

Sangrado y separadores de sentencias (II)

- Las sentencias se terminan al acabarse la línea (salvo casos especiales donde la sentencia queda "abierta": en mitad de expresiones entre paréntesis, corchetes o llaves).
- El caracter \ se utiliza para extender una sentencia más allá de una linea, en los casos en que no queda "abierta".
- El caracter : se utiliza como separador en sentencias compuestas. Ej.: para separar la definición de una función de su código.
- El caracter ; se utiliza como separador de sentencias escritas en la misma línea.

Tuplas

Tipo predefinido de Python para una lista inmutable.
Se define de la misma manera, pero con los elementos entre paréntesis.
Las tuplas no tienen métodos: no se pueden añadir elementos, ni cambiarlos, ni buscar con `index()`.
Sí puede comprobarse la existencia con el operador `in`.

```
>>> tupla = ("a", "b", "mpilgrim", "z", "example")
>>> tupla[0]
'a'
>>> 'a' in tupla
1
>>> tupla[0] =  "b"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Utilidad de las tuplas:

- Son más rápidas que las listas
- Pueden ser una clave de un diccionario (no así las listas)
- Se usan en el formateo de cadenas

`tuple(lista)` devuelve una tupla con los elementos de la lista `lista`
`list(tupla)` devuelve una lista con los elementos de la tupla `tupla`

Funciones predefinidas

- `abs()` valor absoluto
- `float()` convierte a float
- `int()` convierte a int
- `str()` convierte a string
- `round()` redondea
- `raw_input()` acepta un valor desde teclado

Operadores

En orden de precedencia decreciente:

```
+x, -x, ~x    Unary operators
x ** y    Power
x * y, x // y, x % y    Multiplication, division, modulo
x + y, x - y    Addition, subtraction
x << y, x >> y    Bit shifting
x & y    Bitwise and
x | y    Bitwise or
x < y, x <= y, x > y, x >= y, x == y, x != y,
x <> y, x is y, x is not y, x in s, x not in s
                        Comparison, identity,
                        sequence membership tests
not x    Logical negation
x and y    Logical and
lambda args: expr              Anonymous function
```

- La declaración implícita de variables como en perl puede provocar resultados desastrosos

```
#!/usr/bin/perl
$sum_elementos= 3 + 4 + 17;
$media=suma_elementos / 3;    # deletreamos mal la variable
print $media;   # y provocamos resultado incorrecto
```

- Pero Python no permite referenciar variables a las que nunca se ha asignado un valor.

```
#!/usr/bin/python3
sum_elementos = 3 + 4 + 17
media = suma_elementos / 3    # deletreamos mal la variable
print(media)   # y el compilador nos avisa con un error
```

adi

Operaciones sobre cadenas

- `join()` devuelve una cadena que engloba a todos los elementos de la lista.
- `split()` devuelve una lista dividiendo una cadena
- `upper()` devuelve la cadena en mayúsculas
- `lower()` devuelve la cadena en minúsculas

Estas funciones de biblioteca, como todas, podemos encontrarlas en la *python library reference* (disponible en el web en muchos formatos)

Más sobre cadenas

```
#!/usr/bin/python3

cadena = "más vale pájaro en mano"
print(cadena.split())
print(cadena.upper())

otra_cadena = "que,cocodrilo,en,tobillo"
print(otra_cadena.split(','))

lista = ['rojo', 'amarillo', 'verde']
print(lista.join())
```

Operaciones sobre diccionarios

- `len(d)` devuelve el número de elementos de `d`
- `d.has_key(k)` devuelve 1 si existe la clave `k` en `d`, 0 en caso contrario
- `k in d` equivale a: `d.has_key(k)`
- `d.items()` devuelve la lista de elementos de `d`
- `d.keys()` devuelve la lista de claves de `d`

Recogiendo datos del usuario con raw_input

```
#!/usr/bin/python3
entero = int(raw_input("Please enter an integer: "))
if entero < 0:
     entero = 0
     print('Negative changed to zero')
elif entero == 0:
     print('Zero')
elif entero == 1:
     print('Single')
else:
     print('More')
```

No existe `switch/case`

Más sobre listas

- `append()` añade un elemento al final de la lista
- `insert()` inserta un elemento en la posición indicada

```
>>> lista
['a', 'b', 'mpilgrim', 'z', 'example']
>>> lista.append("new")
>>> lista
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> lista.insert(2, "new")
>>> lista
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
```

Más sobre listas

- `index()` busca en la lista un elemento y devuelve el índice de la primera aparición del elemento en la lista. Si no aparece se eleva una excepción.
- El operador `in` devuelve 1 si un elemento aparece en la lista, y 0 en caso contrario.

```
>>> lista
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> lista.index("example")
5
>>> lista.index("new")
2
>>> lista.index("c")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in lista
0
```

Más sobre listas

- `remove()` elimina la primera aparición de un elemento en la lista. Si no aparece, eleva una excepción.
- `pop()` devuelve el último elemento de la lista, y lo elimina. (Pila)
- `pop(0)` devuelve el primer elemento de la lista, y lo elimina. (Cola)

```
 >>> lista
 ['patatas', 'bravas', 'alioli', 'huevo', 'tortilla', 'chorizo']
>>> lista.remove("alioli")
>>> lista
['patatas', 'bravas', 'huevo', 'tortilla', 'chorizo']
>>> lista.remove("oreja")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): orija not in list
>>> lista.pop()
'chorizo'
>>> lista
```

Más sobre listas

- El operador + concatena dos listas, devolviendo una nueva lista
- El operador * concatena repetitivamente una lista a sí misma

```
>>> lista = ['patatas', 'bravas', 'alioli']
>>> lista = lista + ['huevo', 'tortilla']
>>> lista
['patatas', 'bravas', 'alioli', 'huevo', 'tortilla']
>>> lista += ['chorizo']
>>> lista
['patatas', 'bravas', 'alioli', 'huevo', 'tortilla', 'chorizo']
>>> lista = [1, 2] * 3
>>> lista
[1, 2, 1, 2, 1, 2]
```

Más sobre listas

- `sort()` ordena una lista. Puede recibir opcionalmente un argumento especificando una función de comparación, lo que enlentece notable su funcionamiento
- `reverse()` invierte las posiciones de los elementos en una lista.

Ninguno de estos métodos devuelve nada, simplemente alteran la lista sobre la que se aplican.

```
>>> li = ['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'e
>>> li.sort()
>>> li
['a', 'b', 'elements', 'example', 'mpilgrim', 'new', 'new', 'two', 'z']
>>> li.reverse()
>>> li
['z', 'two', 'new', 'new', 'mpilgrim', 'example', 'elements', 'b', 'a']
```

Asignaciones múltiples y rangos

- Pueden hacerse también tuplas de variables:

```
>>> tupla = ('a', 'b', 'e')
>>> (primero, segundo, tercero) = tupla
>>> primero
'a'
```

- La función `range()` permite generar listas al vuelo:

```
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
... FRIDAY, SATURDAY, SUNDAY) = range(7)
>>> MONDAY
0
>>> SUNDAY
6
```

Mapeo de listas

- Se puede mapear una lista en otra, aplicando una función a cada elemento de la lista:

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]
[2, 18, 16, 8]
>>> li
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li]
>>> li
[2, 18, 16, 8]
```

Filtrado de listas

- Sintaxis:

```
[expresión-mapeo for elemento in lista-orig if condición-filtrado]
```

- Ejemplos:

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]
['mpilgrim', 'foo']
```

Control de flujo

Sentencia `while`:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=' ')
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Nótese el efecto de un caracter `,` al final de un `print`
Nótese otro modelo de asignación múltiple

- break sale de un bucle:

```
#!/usr/bin/python3
a=10
while a > 0:
    print(a, end='')
    a=a-1
```

equivale a

```
#!/usr/bin/python3
a=10
while 1:
    print(a, end='')
    if a==1:
        break
    a=a-1
```

Sentencia nula: `pass`
Valor nulo: `None`

Uso de bibliotecas

- Llamada al shell

```
#!/usr/bin/python3
import os
os.system('ls -l')
```

- Argumentos de linea de comandos

```
#!/usr/bin/python3
import sys
print(sys.argv[1:])
```

Las funciones de biblioteca podemos encontrarlas en la *python library reference* (disponible en el web en muchos formatos)

Excepciones

- Un programa sintácticamente correcto puede dar errores de ejecución

```
#!/usr/bin/python3
while 1:
    x=int(raw_input("Introduce un nº"))
    print(x)
```

- Definimos una acción para determinada excepción

```
#!/usr/bin/python3
while 1:
    try:
        x=int(raw_input("Introduce un nº:"))
        print(x)
    except ValueError:
        print("Número incorrecto")
```

- Se puede indicar una acción para cualquier excepción pero es *muy* desaconsejable (enmascara otros errores)
- El programador puede levantar excepciones

```
#!/usr/bin/python3
try:
    x=int(raw_input("Introduce un n°:"))
    print(x)
except :     # para cualquier excepción
    print("Número incorrecto")

raise SystemExit
print("nunca se ejecuta")
```

Objetos en Python

Todo son objetos, en sentido amplio:

- Cualquier objeto puede ser asignado a una variable o pasado como parámetro a una función
- Algunos objetos pueden no tener ni atributos ni métodos
- Algunos objetos pueden no permitir que se herede de ellos

Ejemplos de objetos Python: Strings, listas, funciones, módulos...

Todos los objetos tienen:

- **Identidad**:
  - Nunca cambia.
  - El operador `is` compara la identidad de dos objetos.
  - La función `id()` devuelve una representación de la identidad (actualmente, su dirección de memoria).
- **Tipo**:
  - Nunca cambia.
  - La función `type()` devuelve el tipo de un objeto (que es otro objeto)
- **Valor**:
  - Objetos inmutables: su valor no puede cambiar
  - Objetos mutables: su valor puede cambiar

Contenedores: objetos que contienen referencias a otros objetos (ej.: tuplas, listas, diccionarios).

Cadenas de documentación

- No son obligatorias pero sí muy recomendables (varias herramientas hacen uso de ellas).
- La cadena de documentación de un objeto es su atributo `__doc__`
- En una sola línea para objetos sencillos, en varias para el resto de los casos.
- Entre triples comillas-dobles (incluso si ocupan una línea).
- Si hay varias líneas:
  - La primera línea debe ser una resumen breve del propósito del objeto. Debe empezar con mayúscula y acabar con un punto
  - Una línea en blanco debe separar la primera línea del resto
  - Las siguientes líneas deberían empezar justo debajo de la primera comilla doble de la primera línea

De una sola línea:

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    ...
```

De varias:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
```

Documentando el código (tipo Javadoc)

- Permite documentar el código -generalmente las funciones- dentro del propio código
- Genera la documentación del código en formatos legibles y navegables (HTML, PDF...)
- Se basa en un lenguaje de marcado simple
- PERO... hay que mantener la documentación al día cuando se cambia el código

Ejemplo

```
def interseccion(m, b):
  """
  Devuelve la interseccion de la curva M{y=m*x+b} con el eje X.
  Se trata del punto en el que la curva cruza el eje X (M{y=0}).

  @type  m: número
  @param m: La pendiente de la curva
  @type  b: número
  @param b: La intersección con el eje Y

  @rtype:   número
  @return:  la interseccíoin con el eje X de la curva M{y=m*x+b}
  """
  return -b/m
```

Práctica 1 - Introducción a Python

Protocolos para la Transmisión de Audio y Vídeo en Internet

Gregorio Robles

{grex,jgb}@gsyc.urjc.es
GSyC, Universidad Rey Juan Carlos

30 de septiembre de 2020