prog_datasci_3_apython

October 14, 2020

1 Fundamentos de Programación

1.1 Unidad 3: Estructuras de control y funciones en Python

1.1.1 Instrucciones de uso

Este documento es un *notebook* interactivo que intercala explicaciones más bien teóricas de conceptos de programación con fragmentos de código ejecutables. Para aprovechar las ventajas que aporta este formato, se recomienda, en primer lugar, leer las explicaciones y el código que os proporcionamos. De esta manera tendréis un primer contacto con los conceptos que exponemos. Ahora bien, **¡la lectura es sólo el principio!** Una vez hayáis leído el contenido, no olvidéis ejecutar el código proporcionado y modificarlo para crear variantes que os permitan comprobar que habéis entendido su funcionalidad y explorar los detalles de implementación. Por último, se recomienda también consultar la documentación enlazada para explorar con más profundidad las funcionalidades de los módulos presentados.

1.1.2 Introducción

Esta unidad sigue presentando conceptos básicos de programación en Python. En concreto, presentaremos las operaciones lógicas y veremos cómo alterar el flujo de ejecución de los programas con estructuras iterativas (*for* y *while*), condicionales (*if*) y funciones. Adicionalmente, explicaremos cómo interactuar con archivos de Python, y como importar otros módulos para incorporar funcionalidades adicionales a nuestros programas.

A continuación se incluye la tabla de contenidos, que puede utilizar para navegar por el documento:

6. Bibliografía

2 1. Iteración y operaciones lógicas

En la gran mayoría de ocasiones, tendremos que manipular nuestros datos y para ello utilizaremos los conceptos de iteración y las operaciones lógicas. Las operaciones lógicas nos permiten comparar valores entre variables (mayor, menor, igualdad) y la iteración ir visitando uno a uno los elementos de una lista, tupla, diccionario o cualquier estructura de datos que sea susceptible de secuenciarse.

[1]:
```
# Las operaciones lógicas tendrán como resultado un valor cierto (True) o falso␣
↪(False):

a = 5
b = 1

# ¿Es el valor de a mayor que b?
print(a > b)
```

```
True
```

[2]:
```
# ¿Es el valor de a menor que b?
print(a < b)
```

```
False
```

[3]:
```
b = 5
# ¿Es el valor de b igual al de a?
print(b == a)
```

```
True
```

[4]:
```
# Otros operadores lógicos disponibles son menor o igual '<=', mayor o igual␣
↪'>=' o la negación 'not'.
print(a <= b)
print(a >= b)

a = False
print(not a)
```

```
True
True
True
```

También podemos alterar el flujo de ejecución de nuestro programa utilizando las estructuras *if…else* o *if…elif…else*. Veamos unos cuantos ejemplos:

[5]:
```
a = 5
b = 6
```

```
if a > b:
    print('a es mayor que b')
else:
    print('a es menor o igual que b')
```

```
a es menor o igual que b
```

[6]:
```
a = 5
b = 5
if a > b:
    print('a es mayor que b')
elif a < b:
    print('a es menor que b')
else:
    print('a es igual a b')
```

```
a es igual a b
```

En Python existen solo dos formas de iterar una secuencia: mediante **for** o mediante **while**. La primera de las opciones, *for*, iterará uno por uno los elementos contenidos en una lista. En el caso de *while*, iteraremos mientras la condición de permanencia en el bucle se cumpla. Veamos unos ejemplos:

[7]:
```
monsters = ['Kraken', 'Leviathan', 'Uroborus', 'Hydra']

# Primer método iterando mediante for:
for monster in monsters:
    print(monster)

print()

# Segundo método. La función especial 'enumerate' nos devuelve una tupla en la␣
 ↪que el primer elemento es un
# índice que empieza en 0 y aumenta de 1 en 1 y el segundo elemento es el valor␣
 ↪de la posición en la lista:
for i, monster in enumerate(monsters):
    print(i, monster)
```

```
Kraken
Leviathan
Uroborus
Hydra

0 Kraken
1 Leviathan
2 Uroborus
3 Hydra
```

[8]:
```
# También podríamos iterar la lista mediante while, pero es una forma mucho␣
 ↪menos idiomática en Python
# y preferiremos siempre la opción de for:
i = 0
# Mientras que el índice 'i' sea menor que la longitud de la lista 'monsters':
while i < len(monsters):
    # Imprime el valor de la lista en la posición 'i'.
    print(i, monsters[i])
    # No nos olvidemos de actualizar el valor de 'i' sumándole 1 o tendremos un␣
 ↪bucle infinito.
    i += 1
```

```
0 Kraken
1 Leviathan
2 Uroborus
3 Hydra
```

En este momento seríamos capaces de calcular la serie de Fibonacci hasta un determinado valor:

[9]:
```
# Calculamos el valor de la serie hasta un valor n=100.
n = 100

a, b = 0, 1
while a < n:
    print(a, end = " ")
    a, b = b, a+b
```

```
0 1 1 2 3 5 8 13 21 34 55 89
```

En Python disponemos de una función muy útil para generar una secuencia de números, que podemos utilizar de diferentes formas:

[10]:
```
# Podemos usar la función range para obtener una lista de números:
list(range(10))
```

[10]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Fijaros que `range` no devuelve directamente una lista, sino que devuelve un tipo propio del mismo nombre. Por este motivo, para obtener una lista hay que hacer una conversión de tipo usando `list()`. Esto es una novedad en Python 3, ya que en Python 2 la función `range` devolvía una lista.

[11]:
```
# Visualizamos el tipo de retorno de range
print(type(range(10)))
print(type(list(range(10))))
```

```
<class 'range'>
<class 'list'>
```

Veamos algunos de los usos más habituales de `range`:

[12]:
```
# Podemos utilizar range para iterar:
for i in range(10):
    print(i, end = " ")

print()
```

```
0 1 2 3 4 5 6 7 8 9
```

[13]:
```
# Podemos definir el rango de acción.

# Por ejemplo, especificando únicamente el final como hemos hecho anteriormente
for i in range(10):
    print(i, end = " ")

print()

# especificando inicio y final:
for i in range(5, 10):
    print(i, end = " ")

print()

# o especificar también el salto entre cada valor del resultado:
for i in range(5, 10, 3):
    print(i, end = " ")
```

```
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8
```

Siempre que no necesitemos explícitamente una lista, usaremos directamente el tipo `range` (sin hacer la conversión a lista), ya que esto es en general más eficiente (ahorra memoria). Solo cuando necesitemos una lista, (por ejemplo, para visualizar el resultado como hemos hecho en el primer ejemplo) convertiremos el resultado de `range` a lista.

[14]:
```
# También es posible iterar en un diccionario:
country_codes = {34: 'Spain', 376: 'Andorra', 41: 'Switzerland', 424: None}

# Por clave:
for country_code in country_codes.keys():
    print(country_code)
print()

# Por valor:
for country in country_codes.values():
    print(country)
print()
```

```
# Por los dos a la vez:
for country_code, country in country_codes.items():
    print(country_code, country)
```

```
34
376
41
424

Spain
Andorra
Switzerland
None

34 Spain
376 Andorra
41 Switzerland
424 None
```

3 2. Funciones

Otra forma muy importante de organizar el flujo de ejecución es encapsulando cierta porción de código en una función reutilizable. Una función en Python utiliza el mismo concepto que una función matemática. Por ejemplo, imaginemos la función matemática:

$suma(x, y) = x + y$

En Python podemos definir la misma función de la siguiente forma:

[15]:
```
# La función suma se define mediante la palabra especial 'def' y tiene dos␣
↪argumentos: 'x' e 'y':
def suma(x, y):
    # Devolvemos el valor de la suma.
    return x + y

# En este momento, podemos llamarla con cualquier valor:
print(suma(2, 4))
print(suma(5, -5))
print(suma(3.5, 2.5))
```

```
6
0
6.0
```

[16]:
```
# Podemos definir una función que no haga nada utilizando la palabra especial␣
↪'pass':

def dummy():
```

```
    pass

dummy()
```

[17]:
```
# Podríamos volver a definir el trozo de código de la secuencia de Fibonacci como una función:

def fibonacci(n=100):
    a, b = 0, 1
    while a < n:
        print(a, end = " ")
        a, b = b, a+b

fibonacci(10)
```

```
0 1 1 2 3 5 8
```

[18]:
```
# En el anterior ejemplo, hemos definido que el argumento n tenga un valor por defecto. Esto es muy útil
# en los casos en los que utilicemos la función siempre con un mismo valor, queramos dejar constancia de un
# caso de ejemplo o por defecto. En este caso, podemos ejecutar la función sin pasarle ningún valor:
fibonacci()
```

```
0 1 1 2 3 5 8 13 21 34 55 89
```

[19]:
```
# Podemos definir parte de los argumentos con valores por defecto y otro sin.
# Los argumentos sin valor por defecto siempre tienen que estar más a la izquierda en la definición de la función:

def potencia(a, b=2):
    # Por defecto, elevaremos al cuadrado.
    return a**b

print(potencia(3))
print(potencia(2, 3))
```

```
9
8
```

Recomendamos la lectura de la documentación oficial para acabar de fijar conocimientos.

4 3. Leer y escribir desde ficheros

Una tarea habitual es leer líneas de un fichero o escribir líneas en un fichero. A continuación os explicamos cómo escribir y leer un fichero:

[2]:
```
# Abrimos un fichero de nombre 'a_file.txt' para escritura (de ahí la 'w',␣
↪'writing').
# Lo asignamos a un objeto para manejar el fichero de nombre 'out':
out = open('a_file.txt', 'w')
# Vamos a escribir 10 líneas, cada una con un número de 0 a 9.
for i in range(10):
    # La siguiente línea escribe en el fichero todo lo que pongamos dentro de␣
↪out.write()
    # En nuestro caso, es un string del tipo '0\n', '1\n', etc. Esto lo␣
↪conseguimos
    # utilizando format con un parámetro, que es el número de línea.
    # Añadimos \n al final de cada línea, para incluir el salto de línea.
    out.write("Línea {}\n".format(i))
    # Alternativamente podríamos utilizar las wildcards %d y %s.
    # %s representa un string o cadena de caracteres y %d un número entero.
    # Concatenamos en este caso un número con un string que se trata del salto␣
↪de línea, os.linesep, que equivale en Linux
    # a '\n':
    # out.write("Línea %d%s" % (i, os.linesep))

out.close()
```

[21]:
```
# Ahora vamos a leer el fichero que acabamos de escribir de tres formas␣
↪distintas:

# Primer método
f = open('a_file.txt')
for line in f:
    print(line, end = "")
f.close()
print()

# Segundo método
f = open('a_file.txt')
lines = f.readlines()
for line in lines:
    print(line, end = "")
f.close()
print()

# Tercer método
with open('a_file.txt') as f:
    for line in f:
        print(line, end = "")
```

```
Línea 0
Línea 1
```

```
Línea 2
Línea 3
Línea 4
Línea 5
Línea 6
Línea 7
Línea 8
Línea 9

Línea 0
Línea 1
Línea 2
Línea 3
Línea 4
Línea 5
Línea 6
Línea 7
Línea 8
Línea 9

Línea 0
Línea 1
Línea 2
Línea 3
Línea 4
Línea 5
Línea 6
Línea 7
Línea 8
Línea 9
```

5 4. Organización del código

Un módulo de Python es cualquier fichero con extensión *.py* que esté bajo la ruta del path de Python. El path de Python puede consultarse importando la librería sys:

[22]:
```
import sys
print(sys.path)
```

```
['/usr/lib/python36.zip', '/usr/lib/python3.6', '/usr/lib/python3.6/lib-
dynload', '', '/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-
packages', '/usr/local/lib/python3.6/dist-packages/IPython/extensions',
'/home/datasci/.ipython']
```

Por defecto, Python también mira las librerías que se hayan definido en la variable de entorno $PYTHONPATH (esto puede cambiar ligeramente en un entorno Windows).

Un paquete en Python es cualquier directorio que contenga un fichero especial de nombre *__init__.py* (este fichero estará vacío la mayoría de veces).

Un módulo puede contener diferentes funciones, variables u objetos. Por ejemplo, definamos un módulo de nombre *prog_datasci.py* que contenga:

[23]:
```
# prog_datasci.py

PI = 3.14159265

def suma(x, y):
    return x + y


def resta(x, y):
    return x - y
```

Para utilizar desde otro módulo o script estas funciones, deberíamos escribir lo siguiente:

[24]:
```
from prog_datasci import PI, suma, resta

# Y entonces podríamos utilizarlas normalmente.
siete = suma(2,5)
```

También se suele utilizar la palabra clave `as` para dar un nuevo nombre al módulo o funciones que se importan. Así, por ejemplo, podríamos hacer `from prog_datasci import PI as mypi` para utilizar el nombre `mypi` para referirnos a `PI`:

[3]:
```
from prog_datasci import PI as mypi
print(mypi)
```

```
3.14159265
```

Durante el curso veremos que esta sintaxis es muy común para acortar los nombres de los módulos que importamos.

En Python también podemos utilizar la directiva `from prog_datasci import *`, pero su uso está **totalmente desaconsejado**. La razón es que estaríamos importando gran cantidad de código que no utilizaremos (con el consiguiente aumento del uso de la memoria), pero además, podríamos tener colisiones de nombres (funciones que se llamen de la misma forma en diferentes módulos) sin nuestro conocimiento. A no ser que sea imprescindible, no utilizaremos esa directiva e importaremos una a una las librerías y las funciones que vayamos a necesitar.

Podéis aprender más sobre importar librerías y definir vuestros propios módulos aquí.

6 5. Ejercicios y preguntas teóricas

La parte evaluable de esta unidad consiste en la entrega de un fichero Notebook con extensión «.ipynb» que contendrá los diferentes ejercicios y las preguntas teóricas que hay que contestar. Encontraréis el archivo (`prog_datasci_3_apython_entrega.ipynb`) con las actividades en la misma carpeta que este notebook que estáis leyendo.

6.1 5.1. Instrucciones importantes

Es muy importante que a la hora de entregar el archivo Notebook con vuestras actividades os aseguréis de que:

1. Vuestras soluciones sean originales. Esperamos no detectar copia directa entre estudiantes.
2. Todo el código esté correctamente documentado. El código sin documentar equivaldrá a un 0.
3. El archivo comprimido que entreguéis es correcto (contiene las actividades de la PEC que queréis entregar).

Para hacer la entrega, comprimid el archivo `prog_datasci_3_apython_entrega.ipynb` desde la terminal de la máquina virtual (utilizad vuestro nombre y apellido) y subid el archivo `nombre_apellido_pec3.tgz` al campus virtual:

```
$ cd /home/datasci/prog_datasci_1/prog_datasci_3
$ tar zcvf nombre_apellido_pec3.tgz prog_datasci_3_apython_entrega.ipynb
```

6.2 5.2. Solución

Para descargar la solución de la actividad (a partir de la fecha indicada en el aula), desde la máquina virtual, abrid una terminal y ejecutad el script `get_sol_pec.sh`:

```
datasci@datasci:~$ get_sol_pec.sh
Please type the number of the activity you want to download: 3
Please select the language you want to use: type 0 for Catalan, 1 for Spanish 1
Downloading PEC 3 in SPANISH
Cloning into 'prog_datasci_sol_3'...
remote: Enumerating objects: 26, done.
remote: Counting objects: 100% (26/26), done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 26 (delta 9), reused 0 (delta 0)
Unpacking objects: 100% (26/26), done.
OK! Files downloaded in /home/datasci/prog_datasci_1/prog_datasci_sol_3
```

Una vez descargada la solución, podéis ejecutar el servidor como ya explicamos (mediante el script `start_uoc.sh`) y acceder a su contenido.

7 6. Bibliografía

Os recomendamos que consultéis los siguientes ejemplos del libro *Learn Python 3 the hard way* de la bibliografía de la asignatura, y que intentéis realizar las actividades que se proponen, con el fin de terminar de entender y practicar los conceptos explicados en esta unidad.

Operaciones lógicas: * Exercise 27. Memorizing Logic * Exercise 28. Boolean Practice

Condicionales: * Exercise 29. What If * Exercise 30. Else and If * Exercise 31. Making Decisions

Iteración: * Exercise 32. Loops and Lists * Exercise 33. While-Loops