GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport
ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

UD04. PYTHON (P1)

Sistemas de Gestión Empresarial
2 Curso // CFGS DAM // Informática y Comunicaciones
Alfredo Oltra

Cicles Formatius

ÍNDEX

Versión: 231012.2330

Licencia

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

☑ Atención. Importante prestar atención a esta información.

 Interesante. Ofrece información sobre algun detalle a tener en cuenta.

1 INTRODUCCIÓN

1.1 Python

Python es, junto posiblemente con Rust, uno de los lenguajes que más popularidad ha ganado entre los desarolladores en los últimos años. Aunque no es un lenguaje nuevo y siempre ha tenido muchos seguidores, ha sido la irrupción del tratamiento masivo de datos (*Ciencia de datos e Inteligencia Artificial*) la que lo ha puesto en los primeros puestos de muchos de los índices que cuantifican características como la popularidad en lenguajes de programacion (TIOBE, PYPL).

Pero Python no sólo es, ni mucho menos, un lenguaje para la IA. Es un lenguaje muy versátil y adaptable que permite que tenga cabida en casi todos los aspectos del desarrollo (sistemas, web, IA, multiplataforma...) y en casi todos los niveles. De hecho, para muchos docentes es una de las mejores herramientas para la introducción en el mundo del desarrollo.

1.2 Características

- **Sintaxis clara y legible**. Los programas escritos con Python son auto-expresivos, muy cercanos a un algoritmo escrito en pseudocódigo o lenguaje humano (lenguaje natural). Por ejemplo, evita el uso del punto y coma oi de las llaves o de palabras clave de cierre como endif, utilizando simplemente tabuladores.

  ☑ Aún así no todo es “bonito”. Python permite el uso de una sintáxis específica, un poco más alejada del lenguaje natural, pero que potencia mucho su trabajo. Es la llamada pythonic way.

- **Lenguaje interpretado.** Las instrucciones son traducidas y ejecutadas instrucción a instrucción en tiempo de ejecución lo que facilita el aprendizaje y el desarrollo.
- **Multiplataforma.** El intérprete se encuentra disponible en multitud de plataformas: Linux, MacOS, Windows, iOS, IBM i, Solaris… Esta característica va asociada a la portabilidad: un mismo código puede ser ejecutado sin problemas en cualquiera de estos sistemas.
- **Versátil.** Permite ser utilizado para el desarrollo de multiples campos: desarrollo web (servidor), scripts de automatizaciòn de sistemas, IA, desarrollo de juegos, aplicaciones multiplataforma (con GUI o CLI), aplicaciones para móvil…
- **Potente**. En pocas líneas de código Python puede ejecutar muchas acciones (y habitualmente implementadas de una forma muy óptima) que con otros lenguajes de programación equivaldrían a muchas líneas más para poder conseguir el mismo efecto.
- **Lenguaje no (obligatoriamente) tipado**. El uso más habitual de Python es como lenguaje no tipado, lo cual da mucha libertad y rapidez a la hora de codificar. Esta opción no es siempre bien aceptada por el desarrollador por lo que Python pemite, mediante el uso de anotaciones y herramientas como mypy, gestionar el codigo con tipado fuerte.
- **Gran ecosistema**. Si quieres hacer algo, raro será que no exista en Python una librería o framework que te ayude a ello. Y no sólo la librería: desarrolladores, foros, tutoriales… Además la propia industria lo utiliza de manera masiva, por lo que la inserción laboral es alta.
- **Multiparadigma**. Permite el desarrollo de código mediante programación imperativa, orientado a objetos o funcional.

- **Recolector de basura.** De manera similar a lenguajes como Java o C#, el usuario no tiene que preocuparse por la gestión de la memoria. El propio intérprete es el que detecta aquellos elementos que no son utilizados y los elimina.
- **Curva de aprendizaje suave**. Ya seas un principiante, un desarrollador eventual o un desarrollador avanzado la adaptación a lenguaje resulta sencilla.
- **Libre**. Python se licencia bajo la PSFL (Python Software Foundation License) una licencia compatible con la GPL.

1.3 Inconvenientes

Aunque pueda parecerlo, no todo es bonito en Python. Sus principales inconvenientes son

- Lenguaje interpretado. De la misma que manera que se ha comentado como ventaja, un lenguaje interpretado es lento.
- Existen soluciones alternativas que permiten minimizar esta lentitud, ya sea con otros intérpretes (como IronPython) o con conversores a otros lenguajes compilados, como cython.
- Alto consumo de memoria
- Limitado en el desarrollo para móviles y web, especialmente en el front.

2 ELEMENTOS DEL LENGUAJE

2.1 Comentarios

Los comentarios se realizan con el carácter # para una línea y tres comillas (simples o dobles) para multilínea. Además, las 3 comillas pueden utilizarse para definir cadenas multilínea.

```
# Comentarios de una línea comienzan con una almohadilla (o sostenido)

"""
Strings multilinea pueden escribirse   usando tres "'s, y
comúnmente son usados como comentarios.
"""
```

2.2 Otipos de datos y operadores

```
####################################################
## 1. Tipos de datos primitivos y operadores.
####################################################
```

La nomenclatura para esta sección es *operación # => resultado esperado*, donde la parte a la derecha del carácter # es un comentario y solo nos da información de como va a funcionar la operación.

Si en *Python* pones un número, obtienes simplemente ese número.

```
# Tienes números
3 # => 3
```

Si realizas operaciones aritméticas con enteros, obtienes el resultado con un número entero. Los paréntesis modifican la precedencia entre operadores.

```
# Lo que esperarías
1 + 1 # => 2
8 - 1 # => 7
10 * 2 # => 20
# Refuerza la precedencia con paréntesis
(1 + 3) * 2  # => 8
```

La división, aunque entre enteros devuelve un tipo de dato *float* (decimal) si se hace con `/`, pero si se desea un resultado entero (con truncado de decimales) puedes utilizar `//`.

```
# La división por defecto retorna un número 'float' (número de coma
flotante)
35 / 5  # => 7.0
# Sin embargo también tienes disponible división entera
34 // 5 # => 6
```

Si en una operación aritmética, alguno de los dos operadores es un *float*, el resultado siempre es un *float* (se convierte al tipo de datos que engloba mayor número datos).

```
# Cuando usas un float, los resultados son floats
3 * 2.0 # => 6.0
```

Aquí vemos el tipo de datos *boolean* y los operadores lógicos que nos devuelven un *boolean*.

```
# Valores 'boolean' (booleanos) son tipos primitivos
True
False

# Niega con 'not'
not True  # => False
not False  # => True

# Igualdad es ==
1 == 1  # => True
2 == 1  # => False

# Desigualdad es !=
1 != 1  # => False
2 != 1  # => True

# Más comparaciones
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# ¡Las comparaciones pueden ser concatenadas!
1 < 2 < 3  # => True
```

```
2 < 3 < 2  # => False
'SGE' == 'SGE' != 'ERP'  # => True
```

Aquí observamos como definir *Strings* (cadenas de caracteres) y como operar con ellos (formato, concatenación, acceso a un elemento, etc.)

```
# Strings se crean con comillas dobles o simples (?)
"Esto es un string."
'Esto también es un string'

# ¡Strings también pueden ser sumados!
"Hola " + "mundo!" # => "Hola mundo!"

# Pueden ser concatenados mediante una cadena
"*".join(["dato1","dato2","dato3"])  # => "dato1*dato2*dato3"

# Un string puede ser tratado como una lista de caracteres
"Esto es un string"[0]   'E'

# .format puede ser usaro para darle formato a los strings, así:
"{} pueden ser {}".format("strings", "interpolados")

# Puedes reutilizar los argumentos de formato si estos se repiten.
"{0} sé ligero, {0} sé rápido, {0} brinca sobre la {1}".format("Jack",
"vela") # => "Jack sé ligero, Jack sé rápido, Jack brinca sobre la vela"
# Puedes usar palabras claves si no quieres contar.
"{nombre} quiere comer {comida}".format(nombre="Bob", comida="lasaña") #
=> "Bob quiere comer lasaña"
# También puedes interpolar cadenas usando variables en el contexto
nombre = 'Bob'
comida = 'Lasaña'
f'{nombre} quiere comer {comida}'  # => "Bob quiere comer lasaña"
```

None es un objeto predefinido en *Python*, utilizado para comparar si algo es “nada”.

```
# None es un objeto
None  # => None

# No uses el símbolo de igualdad `==` para comparar objetos con None
# Usa `is` en su lugar
"etc" is None # => False
None is None  # => True

# None, 0, y strings/listas/diccionarios/conjuntos vacíos(as) todos se
evalúan como False.
# Todos los otros valores son True
bool(0)  # => False
bool("")  # => False
bool([]) # => False
bool({}) # => False
bool(set()) # => False
```

2.3 Variables y colecciones

```
####################################################
## 2. Variables y Colecciones
####################################################
```

La función print nos permite imprimir cadenas de caracteres.

```
# Python tiene una función para imprimir
print("Soy Python. Encantado de conocerte")
```

En *Python* no es necesario declarar variables antes de utilizarlas. Una convención habitual es usar _ para separar las palabras (Snake Case), pero hay otras como Camel Case.

```
# No hay necesidad de declarar las variables antes de asignarlas.
una_variable = 5    # La convención es usar guiones_bajos_con_minúsculas
una_variable # => 5
otraVariable = 3    # Aquí en formato Camel Case
otraVariable # => 3
# Acceder a variables no asignadas previamente es una excepción.
```

```
# Ver Control de Flujo para aprender más sobre el manejo de excepciones.
otra_variable  # Levanta un error de nombre
```

La principal colección de elementos en *Python* son las listas. Aquí vemos ejemplos de uso:

```
# Listas almacena secuencias
lista = []
# Puedes empezar con una lista prellenada
otra_lista = [4, 5, 6]

# Añadir cosas al final de una lista con 'append'
lista.append(1)     #lista ahora es [1]
lista.append(2)     #lista ahora es [1, 2]
lista.append(4)     #lista ahora es [1, 2, 4]
lista.append(3)     #lista ahora es [1, 2, 4, 3]

# Remueve del final de la lista con 'pop'
lista.pop()         # => 3 y lista ahora es [1, 2, 4]

# Pongámoslo de vuelta
lista.append(3)     # Nuevamente lista ahora es [1, 2, 4, 3].
```

Para acceder a elementos de una lista, accedemos como accederemos en otros lenguajes a un array: si tiene N elementos, con valores del 0 al N-1. Además *Python* permite referencia negativas. Por ejemplo, -1 en una lista de N elementos, equivale a acceder al elemento “N-1”.

```
# Accede a una lista como lo harías con cualquier arreglo
lista[0] # => 1

# Mira el último elemento
lista[-1] # => 3

# Mirar fuera de los límites es un error 'IndexError'
lista[4] # Levanta la excepción IndexError
```

Las listas permite obtener una nueva lista formada por un rango de elementos usando *:*. La parte izquierda al *:* es donde comienza, y la parte derecha donde acaba. Si se mete un segundo “:”, indica el número de pasos del rango a tomar. Al final sigue una sintaxis *lista[inicio:final:pasos].*

A continuación, algunos ejemplos de rangos y otras operaciones (concatenación, comprobar elementos, tamaño, borrado, etc.) con listas:

```
# Puedes mirar por rango con la sintáxis de trozo.
# (Es un rango cerrado/abierto para los matemáticos.)
lista[1:3] # => [2, 4]
# Omite el inicio
lista[2:] # => [4, 3]
# Omite el final
lista[:3] # => [1, 2, 4]
# Selecciona cada dos elementos
lista[::2]   # =>[1, 4]
# Invierte la lista
lista[::-1]   # => [3, 4, 2, 1]
# Usa cualquier combinación de estos para crear trozos avanzados
# lista[inicio:final:pasos]

# Remueve elementos arbitrarios de una lista con 'del'
del lista[2] # lista ahora es [1, 2, 3]

# Puedes sumar listas
lista + otra_lista # => [1, 2, 3, 4, 5, 6] - Nota: lista y otra_lista no
se tocan

# Concatenar listas con 'extend'
lista.extend(otra_lista) # lista ahora es [1, 2, 3, 4, 5, 6]

# Verifica la existencia en una lista con 'in'
1 in lista # => True

# comprueba que al menos uno de los elementos de la lista (puede ser
cualquier iterable) sea cierto
any(['True', 'False', 'True']) => True
any([0, 'False']) => False

# Examina el largo de una lista con 'len'
len(lista) # => 6
```

Una forma interesante de crear listas es la llamada *List comprehension*, que permite la creación de listas a partir de otro iterable de una manera muy compacta.

```
lista = [5, 12, 3, 45]
lista_mayores_10 = [ x for x in lista if x > 10]
```

Otro elemento (menos utilizado en *Python* que las listas) son las tuplas. Las tuplas son como las listas, solo que son inmutables (no podemos cambiar valores, añadir, borrar, etc.).

```
# Tuplas son como listas pero son inmutables.
tupla = (1, 2, 3)
tupla[0] # => 1
tupla[0] = 3  # Levanta un error TypeError

# También puedes hacer todas esas cosas que haces con listas
len(tupla) # => 3
tupla + (4, 5, 6) # => (1, 2, 3, 4, 5, 6)
tupla[:2] # => (1, 2)
2 in tupla # => True

# Puedes desempacar tuplas (o listas) en variables
a, b, c = (1, 2, 3)     # a ahora es 1, b ahora es 2 y c ahora es 3
# Tuplas son creadas por defecto si omites los paréntesis
d, e, f = 4, 5, 6

# Es muy fácil es intercambiar dos valores
e, d = d, e    # d ahora es 5 y e ahora es 4
```

Otra estructura de datos interesante y óptima es la implementación de *diccionarios* (es decir, asociación clave/valor) en *Python* mediante la estructura *{ }*. A continuación vemos algunos ejemplos.

```
# Diccionarios relacionan claves y valores
dicc_vacio = {}

# Aquí está un diccionario pre-rellenado
dicc_lleno = {"uno": 1, "dos": 2, "tres": 3}

# Busca valores con []
dicc_lleno["uno"] # => 1

# Obtén todas las claves como una lista con 'keys()'. Necesitamos
envolver la llamada en 'list()' porque obtenemos un iterable.
list(dicc_lleno.keys()) # => ["tres", "dos", "uno"]
# Nota - El orden de las claves del diccionario no está garantizada.
```

```python
# Tus resultados podrían no ser los mismos del ejemplo.

# Obtén todos los valores como una lista. Nuevamente necesitamos
envolverlas en una lista para sacarlas del iterable.
list(dicc_lleno.values()) # => [3, 2, 1]
# Nota - Lo mismo que con las claves, no se garantiza el orden.

# Verifica la existencia de una llave en el diccionario con 'in'
"uno" in dicc_lleno # => True
1 in dicc_lleno # => False

# Buscar una llave inexistente deriva en KeyError
dicc_lleno["cuatro"] # KeyError

# Usa el método 'get' para evitar la excepción KeyError
dicc_lleno.get("uno") # => 1
dicc_lleno.get("cuatro") # => None

# El método 'get' soporta un argumento por defecto cuando el valor no
existe.
dicc_lleno.get("uno", 4) # => 1
dicc_lleno.get("cuatro", 4) # => 4

# El método 'setdefault' inserta en un diccionario solo si la llave no
está presente
dicc_lleno.setdefault("cinco", 5) #dicc_lleno["cinco"] es puesto con
valor 5

dicc_lleno.setdefault("cinco", 6) #dicc_lleno["cinco"] todavía es 5

# Elimina claves de un diccionario con 'del'
del dicc_lleno['uno'] # Remueve la llave 'uno' de dicc_lleno
```

Otra estructura de datos óptima para este proceso son los conjuntos. Permite hacer de manera

```python
# Sets (conjuntos) almacenan conjuntos
conjunto_vacio = set()

# Inicializar un conjunto con montón de valores.
un_conjunto = {1,2,2,3,4} # un_conjunto ahora es {1, 2, 3, 4}

# Añade más valores a un conjunto
conjunto_lleno.add(5) # conjunto_lleno ahora es {1, 2, 3, 4, 5}

# Haz intersección de conjuntos con &
otro_conjunto = {3, 4, 5, 6}
conjunto_lleno & otro_conjunto # => {3, 4, 5}

# Haz unión de conjuntos con |
conjunto_lleno | otro_conjunto # => {1, 2, 3, 4, 5, 6}

# Haz diferencia de conjuntos con -
{1,2,3,4} - {2,3,5} # => {1, 4}

# Verifica la existencia en un conjunto con 'in'
2 in conjunto_lleno # => True
10 in conjunto_lleno # => False
```

2.4 Control de flujo

```
####################################################
## 3. Control de Flujo
####################################################
```

Aquí vemos ejemplos de como utilizar la estructura de control de flujo `if`:

```
# Creemos una variable para experimentar
some_var = 5

# Aquí está una declaración de un 'if'. ¡La indentación es significativa
en Python!
# imprime "una_variable es menor que 10"
if una_variable > 10:
    print("una_variable es más grande que 10.")
elif una_variable < 10:  # Este condición 'elif' es opcional.
    print("una_variable es más pequeña que 10.")
else: # Esto también es opcional.
    print("una_variable vale 10.")
```

Aquí vemos como utilizar la estructura `for` para iterar sobre cada elemento de los elementos que *Python* considera *iterables* (listas, tuplas, diccionarios, etc.).

```
"""
For itera sobre iterables (listas, cadenas, diccionarios, tuplas,
generadores...)
imprime:
    perro es un mamifero
    gato es un mamifero
    raton es un mamifero
"""
for animal in ["perro", "gato", "raton"]:
    print("{} es un mamífero".format(animal))
```

La función `range` es un generador de números. Nos puede ayudar para realizar iteraciones numéricas utilizando `for`:

```
"""
range(número inicio, número fin, incremento) retorna un generador de
números
desde número inicio (por defecto 0) hasta el número fin con un
incremento de valor incremento (por defecto 1)
El único requerido es el número fin
imprime:
     0
     1
     2
     3
"""
for i in range(4):
     print(i)
```

La estructura de control de flujo `while` itera mientras una condición sea cierta.

```
"""
While itera hasta que una condición no se cumple.
imprime:
     0
     1
     2
     3
"""
x = 0
while x < 4:
     print(x)
     x += 1  # versión corta de x = x + 1
```

Python 3 permite el manejo de excepciones mediante `try` y `catch` como se observa aquí:

```
# Maneja excepciones con un bloque try/except
try:
    # Usa raise para levantar un error
    raise IndexError("Este es un error de indice")
except IndexError as e:
    pass # Pass no hace nada (“pasa”). Usualmente aquí harias alguna recuperacion.
```

Aquí vemos un ejemplo de como crear elementos iterables y algunas propiedades. En el ejemplo, trabajaremos utilizando las claves (keys) de un diccionario y poder recorrerlos con un `for`.

3 BIBLIOGRAFIA

1. Learn X in Y Minutes.
2. Aprende Python con Alf
3. Python para todos.

4 AUTORES

A continuación ofrecemos en orden alfabético (por apellido) el listado de autores que han hecho aportaciones a este documento.

- Jose Castillo Aliaga
- Sergi García Barea
- Alfredo Oltra