Sistemes de Gestió Empresarial

UD05 Python 3 bàsic

Actualitzat Octubre 2021

Llicència

Nomenclatura

Al llarg d'aquest tema s'utilitzaran diferents símbols per distingir elements importants dins del contingut. Aquests símbols són:

> 📖 **Important**

> ❕ **Atenció**

> 💬 **Interessant**

ÍNDEX DE CONTINGUT

1. INTRODUCCIÓ

En aquest document, realitzarem un resum dels principals elements del llenguatge Python, basant-nos en la documentació de https://learnxinyminutes.com/docs/es-es/python-es/, comentant cada element i en alguns casos afegint exemples addicionals.

2. RESUM CARACTERÍSTIQUES PYTHON 3

Per allò que respecta Python 3, comentar que es tracta d'una opció molt atractiva per començar a codificar o per aprendre ràpidament per a programadors experts. Algunes característiques:

- **Sintaxis senzilla**: els programes escrits amb Python 3 són autoexpressius, molts cerca'ns a un algoritme escrit en pseudocodi o llenguatge natural.
- **Molt potent**: en poques línies de codi, Python pot executar moltes accions (i habitualment implementades d'una forma òptima) que amb altres llenguatges de programació equivaldrien a moles més línies per a poder aconseguir el mateix efecte.
- **Llenguatge interpretat**: les instruccions són traduïdes i executades instrucció a instrucció. No hi ha fitxers de codi intermedi, ni tampoc temps de compilació.
- **Llenguatge sense obligació de declarar tipus de dades**: aquest aspecte pot considerar-se un avantatge o un inconvenient dependent de desenvolupador de codi.
- **Corba d'aprenentatge suau**: per començar a programar des de zero, Python és de les millors opcions. A més, per a programadors experts és un llenguatge senzill d'aprendre.

3. ELEMENTS DEL LLENGUATGE PYTHON 3

3.1 Comentari

Els comentaris es realitzen amb el caràcter "#" per a una línia i "tres cometes" per a multilínia. A més, les 3 cometes poden utilitzar-se per a definir cadenes multilínia.

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

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

3.2 Tipus de dades i operadors

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

La nomenclatura para aquesta secció és “operació # => resultat esperat”, on la part a la dreta del caràcter “#” es un comentari i només ens dona informació de com va a funcionar l’operació.

Si en Python 3 poses un número, obtens simplement eixe número.

```
# Tienes números
3 # => 3
```

Si realitzes operacions aritmètiques amb enters, obtens el resultat amb un nombre enter. Els parèntesis modifiquen la precedència entre operadors.

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

La divisió, encara que entre enters, retorna un tipus de dada “float” (decimal) si es fa amb “/”, però si es desitja un resultat sencer (amb truncat de decimals) pots utilitzar “//”.

```
# Excepto la división la cual 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 operació aritmètica, algun dels dos operadors és un “float”, el resultat sempre és un float (es converteix a la mena de dades que major engloba).

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

Ací veiem el tipus de dades “boolean” i els operadors lògics que ens retornen 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
```

Ací observem com definir “Strings” (cadenes de caràcters) i com operar amb ells (format, concatenació, accés a un element, etc.)

```
# Strings se crean con " o '
"Esto es un string."
'Esto también es un string'

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

# 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 és un objecte predefinit en Python, utilitzat per a comparar si alguna cosa és “res”.

```
# 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
```

3.3 Variables i coleccions

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

La funció “print” ens permet imprimir cadenes de caràcters.

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

En Python no és necessari declarar variables abans d'utilitzar-les. Una convenció és usar “_” per a separar les paraules, però hi ha altres com Camel Case https://es.wikipedia.org/wiki/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      # Aqui en formato Camel Case
otraVariable # => 3
```

```
# Acceder a variables no asignadas previamente es una excepción.
# Ve Control de Flujo para aprender más sobre el manejo de excepciones.
otra_variable  # Levanta un error de nombre
```

La principal col·lecció d'elements en Python són les llistes. Ací veiem exemples d'ús:

```
# 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].
```

Per a accedir a elements d'una llista, accedim com accedirem en altres llenguatges a un array: si té N elements, amb valors del 0 al N-1. A més Python permet referència negatives. Per exemple, -1 en una llista de N elements, equival a accedir a l'element “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
```

Les llistes permet obtenir una nova llista formada per un rang d'elements usant “:”. La part esquerra al “:” és on comença, i la part dreta on acaba. Si es fica un segon “:”, indica el nombre de passos del rang a prendre. Al final segueix una sintaxi “llista[inici:final:passos]”.

A continuació, alguns exemples de rangs i altres operacions (concatenació, comprovar elements, grandària, esborrat, etc.) amb llistes:

```
# 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

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

Un altre element (menys utilitzat en Python que les llistes) són les tuplas. Les tuplas són com les llistes, només que són immutables (no podem canviar valors, afegir, esborrar, 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
# Ahora mira que fácil es intercambiar dos valores
e, d = d, e      # d ahora es 5 y e ahora es 4
```

Una altra estructura de dades interessant i òptima és la implementació de diccionaris (és a dir, associació clau/valor) en Python mitjançant l'estructura "{ }". A continuació veiem alguns exemples.

```
# 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. Hablaremos
de eso luego.
list(dicc_lleno.keys()) # => ["tres", "dos", "uno"]
# Nota - El orden de las claves del diccionario no está garantizada.
# 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
```

Una altra estructura de dades òptima per a aquest procés són els conjunts. Permet fer de manera òptima operacions relacionades amb els conjunts (intersecció, unió, etc.).

```
# Sets (conjuntos) almacenan conjuntos
conjunto_vacio = set()
# Inicializar un conjunto con montón de valores. Yeah, se ve un poco
como un diccionario. Lo siento.
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
```

3.4 Control de flux

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

Ací veiem exemples de com utilitzar l'estructura de control de flux “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 completamente mas grande que 10.")
elif una_variable < 10:    # Este condición 'elif' es opcional.
      print("una_variable es mas chica que 10.")
else:           # Esto también es opcional.
      print("una_variable es de hecho 10.")
```

Ací veiem com utilitzar l'estructura “for” per a iterar sobre cada element dels elements que Python considera “iterables” (llestes, tuplas, diccionaris, 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 mamifero".format(animal))
```

La funció “range” és un generador de números. Ens pot ajudar per a realitzar iteracions numèriques utilitzant for:

```
"""
`range(número)` retorna un generador de números
desde cero hasta el número dado
imprime:
     0
     1
     2
     3
"""
for i in range(4):
     print(i)
```

L'estructura de control de flux “While”, itera mentre una condició siga certa.

```
"""
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 permet el maneig d'excepcions mitjançant “try” i “catch” com s'observa ací:

```
# 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.
```

Ací veiem un exemple de com crear elements iterables i algunes propietats. En l'exemple, treballarem utilitzant les “claus” (keys) d'un diccionari i poder recórrer-los amb un for.

```
# Python ofrece una abstracción fundamental llamada Iterable.
# Un iterable es un objeto que puede ser tratado como una sequencia.
# El objeto es retornado por la función 'range' es un iterable.

dicc_lleno = {"uno": 1, "dos": 2, "tres": 3}
nuestro_iterable = dicc_lleno.keys()
print(nuestro_iterable) # => dict_keys(['uno', 'dos', 'tres']). Este es
un objeto que implementa nuestra interfaz Iterable

Podemos recorrerla.
for i in nuestro_iterable:
     print(i)   # Imprime uno, dos, tres

# Aunque no podemos selecionar un elemento por su índice.
nuestro_iterable[1]  # Genera un TypeError

# Un iterable es un objeto que sabe como crear un iterador.
nuestro_iterator = iter(nuestro_iterable)

# Nuestro iterador es un objeto que puede recordar el estado mientras lo
recorremos.
# Obtenemos el siguiente objeto llamando la función __next__.
nuestro_iterator.__next__()  # => "uno"

# Mantiene el estado mientras llamamos __next__.
nuestro_iterator.__next__()  # => "dos"
nuestro_iterator.__next__()  # => "tres"

# Después que el iterador ha retornado todos sus datos, da una excepción
StopIterator.
nuestro_iterator.__next__() # Genera StopIteration

# Puedes obtener todos los elementos de un iterador llamando a list() en
el.
list(dicc_lleno.keys())  # => Retorna ["uno", "dos", "tres"]
```

3.5 Funcions

```
####################################################
## 4. Funciones
####################################################
```

Ací alguns exemples de definició i crida de funcions en Python 3.

```
# Usa 'def' para crear nuevas funciones
def add(x, y):
    print("x es {} y y es {}".format(x, y))
    return x + y    # Retorna valores con una la declaración return

# Llamando funciones con parámetros
add(5, 6) # => imprime "x es 5 y y es 6" y retorna 11

# Otra forma de llamar funciones es con argumentos de palabras claves
add(y=6, x=5)   # Argumentos de palabra clave pueden ir en cualquier
orden.


# Puedes definir funciones que tomen un número variable de argumentos
def varargs(*args):
    return args

varargs(1, 2, 3) # => (1,2,3)


# Puedes definir funciones que toman un número variable de argumentos
# de palabras claves
def keyword_args(**kwargs):
    return kwargs

# Llamémosla para ver que sucede
keyword_args(pie="grande", lago="ness") # => {"pie": "grande", "lago":
"ness"}


# Puedes hacer ambas a la vez si quieres
def todos_los_argumentos(*args, **kwargs):
```

```
    print args
    print kwargs
"""
todos_los_argumentos(1, 2, a=3, b=4) imprime:
    (1, 2)
    {"a": 3, "b": 4}
"""

# ¡Cuando llames funciones, puedes hacer lo opuesto a varargs/kwargs!
# Usa * para expandir tuplas y usa ** para expandir argumentos de
palabras claves.
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
todos_los_argumentos(*args) # es equivalente a foo(1, 2, 3, 4)
todos_los_argumentos(**kwargs) # es equivalente a foo(a=3, b=4)
todos_los_argumentos(*args, **kwargs) # es equivalente a foo(1, 2, 3, 4,
a=3, b=4)
```

Per a facilitar algunes operacions, Python permet tant funcions definides (de primera classe) com a funcions anònimes. Aquestes funcions anònimes ens ajuden sobretot a utilitzar "programació funcional" amb funcions com "map", "filter" i "reduce".

```
# Python tiene funciones de primera clase
def crear_suma(x):
    def suma(y):
    return x + y
    return suma

sumar_10 = crear_suma(10)
sumar_10(3) # => 13

# También hay funciones anónimas
(lambda x: x > 2)(3) # => True

# Hay funciones integradas de orden superior
map(sumar_10, [1,2,3]) # => [11, 12, 13]
filter(lambda x: x > 5, [3, 4, 5, 6, 7]) # => [6, 7]

# Podemos usar listas por comprensión para mapeos y filtros agradables
[add_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
```

```
[x for x in [3, 4, 5, 6, 7] if x > 5] # => [6, 7]
# también hay diccionarios
{k:k**2 for k in range(3)} # => {0: 0, 1: 1, 2: 4}
# y conjuntos por comprensión
{c for c in "la cadena"} # => {'d', 'l', 'a', 'n', ' ', 'c', 'e'}
```

3.6 Classes

```
#####################################################
## 5. Clases
#####################################################
```

Les classes en Python, hereten inicialment de l'objecte predefinit “object”. Ací un exemple de definició de classe amb atributs, constructor i mètodes.

```
# Heredamos de object para obtener una clase.
class Humano(object):

    # Un atributo de clase es compartido por todas las instancias de
esta clase
    especie = "H. sapiens"

    # Constructor basico
    def __init__(self, nombre):
    # Asigna el argumento al atributo nombre de la instancia
        self.nombre = nombre

    # Un metodo de instancia. Todos los metodos toman self como primer
argumento
    def decir(self, msg):
        return "%s: %s" % (self.nombre, msg)

    # Un metodo de clase es compartido a través de todas las
instancias
    # Son llamados con la clase como primer argumento
    @classmethod
    def get_especie(cls):
        return cls.especie

    # Un metodo estatico es llamado sin la clase o instancia como
referencia
```

```
    @staticmethod
    def roncar():
        return "*roncar*"


# Instancia una clase
i = Humano(nombre="Ian")
print i.decir("hi")   # imprime "Ian: hi"

j = Humano("Joel")
print j.decir("hello")  #imprime "Joel: hello"

# Llama nuestro método de clase
i.get_especie() # => "H. sapiens"

# Cambia los atributos compartidos
Humano.especie = "H. neanderthalensis"
i.get_especie() # => "H. neanderthalensis"
j.get_especie() # => "H. neanderthalensis"

# Llama al método estático
Humano.roncar() # => "*roncar*"
```

3.7 Móduls

```
####################################################
## 6. Módulos
####################################################
```

Python permet importar mòduls, tant creats per nosaltres, com a existents en el sistema. Una potent eina per a descarregar mòduls més populars és “pip” https://pypi.org/project/pip/

```
# Puedes importar módulos
import math
print(math.sqrt(16)) # => 4.0

# Puedes obtener funciones específicas desde un módulo
```

```
from math import ceil, floor
print(ceil(3.7)) # => 4.0
print(floor(3.7))# => 3.0

# Puedes importar todas las funciones de un módulo
# Precaución: Esto no es recomendable
from math import *

# Puedes acortar los nombres de los módulos
import math as m
math.sqrt(16) == m.sqrt(16) # => True

# Los módulos de Python son sólo archivos ordinarios de Python.
# Puedes escribir tus propios módulos e importarlos. El nombre del
módulo
# es el mismo del nombre del archivo.

# Puedes encontrar que funciones y atributos definen un módulo.
import math
dir(math)
```

3.8 Avanzat: generadors i decoradors

Exemple de creació de generadors:

```
# Los generadores te ayudan a hacer un código perezoso (lazy)
def duplicar_numeros(iterable):
     for i in iterable:
          yield i + i

# Un generador crea valores sobre la marcha.
# En vez de generar y retornar todos los valores de una vez, crea uno en
cada iteración.
# Esto significa que valores más grandes que 15 no serán procesados en
'duplicar_numeros'.
# Fíjate que 'range' es un generador. Crear una lista 1-900000000
tomaría mucho tiempo en crearse.
_rango = range(1, 900000000)
# Duplicará todos los números hasta que un resultado >= se encuentre.
for i in duplicar_numeros(_rango):
```

```
        print(i)
        if i >= 30:
            break
```

Exemple d'utilització de decoradors en Python.

```
# Decoradores
# en este ejemplo 'pedir' envuelve a 'decir'
# Pedir llamará a 'decir'. Si decir_por_favor es True entonces cambiará
el mensaje a retornar
from functools import wraps


def pedir(_decir):
    @wraps(_decir)
    def wrapper(*args, **kwargs):
        mensaje, decir_por_favor = _decir(*args, **kwargs)
        if decir_por_favor:
            return "{} {}".format(mensaje, "¡Por favor! Soy pobre
:(")
        return mensaje

    return wrapper

@pedir
def say(decir_por_favor=False):
    mensaje = "¿Puedes comprarme una cerveza?"
    return mensaje, decir_por_favor


print(decir())  # ¿Puedes comprarme una cerveza?
print(decir(decir_por_favor=True))  # ¿Puedes comprarme una cerveza?
¡Por favor! Soy pobre :(
```

4. BIBLIOGRAFÍA

- Learn X in Y Minutes: https://learnxinyminutes.com/docs/es-es/python-es/
- Aprende Python con Alf https://aprendeconalf.es/docencia/python/manual/
- Python para todos: http://do1.dr-chuck.com/pythonlearn/ES_es/pythonlearn.pdf

5. AUTORS (EN ORDRE ALFABÈTIC)

A continuació oferim en ordre alfabètic el llistat d'autors que han fet aportacions a aquest document.

- Jose Castillo Aliaga
- Sergi García Barea

Gran part del contingut ha sigut obtingut del material amb llicència CC BY SA disponible enhttps://learnxinyminutes.com/docs/es-es/python-es/