PROGRAMACIÓN AVANZADA CON PYTHON

(CEFIRE CTEM)

Repaso de conceptos

Autora: María Paz Segura Valero (segura_marval@gva.es)

CONTENIDO

1. Introducción....2
2. Qué es Python....2
    1.1. Versiones de Python....2
3. Cómo trabajar con Python....2
4. Algunos conceptos básicos....3
5. Funciones integradas....5
    5.1. Funciones de entrada y salida....5
6. Módulos en Python....6
7. Estructuras de control....7
    7.1. Instrucciones alternativas: if, else, elif....7
    7.2. Instrucciones repetitivas: while, for....8
    7.3. Otras instrucciones: break, continue, pass....9
8. Definición de funciones....9
9. Tipos de datos complejos....10
    9.1. Tipo de datos LISTA....10
    9.2. Tipo de datos DICCIONARIO....12
10. Excepciones....14
11. Fuentes de información....15

1. Introducción

Si estás leyendo este documento es porque ya tienes nociones básicas de Python y has decidido aprender **conceptos más avanzados**.

No obstante, la primera sesión la utilizaremos para hacer un **repaso general** de los conceptos básicos necesarios para trabajar con Python. Si necesitas que aclare algún concepto o aporte documentación extra puedes solicitarlo a través de los foros del curso.

¡Empezamos!

2. Qué es Python

Python es un lenguaje de programación que fue creado a finales de los ochenta por **Guido van Rossum** en el *Centro para las Matemáticas y la Informática* (CWI, Centrum Wiskunde & Informatica) de los Países Bajos.

Guido es un gran admirador del grupo humorístico británico **Monty Python** y decidió bautizar al nuevo lenguaje inspirándose en el nombre de dicho grupo.

Según su página oficial (https://www.python.org/), "Python es un lenguaje de programación que te permite trabajar rápido e integrar sistemas de una manera más efectiva".

1.1. Versiones de Python

A día de hoy disponemos de la versión 3.9.6 de Python tanto para Linux/Unix como para Windows o Mac OS. Este documento está basado en la versión 3.8.5.

En su página oficial (https://www.python.org/downloads/) podemos encontrar esta versión e incluso versiones anteriores o versiones específicas para otras plataformas como AIX, IBM i, iOS, iPadOS, Solaris, etc .

3. Cómo trabajar con Python

Existen multitud de formas para crear y probar programas en Python. Desde utilizar el intérprete de comandos que incorpora la instalación básica del lenguaje hasta usar plataformas on-line donde podemos olvidarnos completamente de cualquier tarea de instalación.

Cada opción tiene sus ventajas e inconvenientes, así que la elección de una u otra dependerá de tus necesidades y recursos.

No obstante, utilices la opción que utilices, debes saber que todos los programas de Python se almacenan en ficheros con extensión ***.py***

Nuestra recomendación es que utilices el **IDE Thonny**, un entorno de desarrollo integrado que incorpora un depurador para detectar errores en la prueba de programas. En el aula virtual puedes encontrar un documento específico que explica cómo instalar y trabajar con el *IDE Thonny*.

4. Algunos conceptos básicos

En Python las variables no hace falta declararlas antes de ser utilizadas ya que es en el momento de ejecución del programa cuando se conoce el tipo de dato que guarda cada variable y los operadores y funciones que se le pueden aplicar. A este mecanismo se le conoce como **tipado dinámico de variables**.

En Python **todos los elementos son objetos**. Es decir, una variable de tipo entero será una variable que referenciará a un objeto de tipo *entero* o una variable de tipo cadena de texto será una variable que referenciará a un objeto de tipo *cadena de texto*.

En la siguiente tabla puedes ver los tipos de datos más usuales en Python:

Clasificación	Clase en Python	Descripción
Números	int	Número entero. Ejemplos: 3, 55, 1948.
	float	Número real. Ejemplos: 10.3348, .98344, 193., 1.9.
	complex	Número complejo con parte real y parte imaginaria. Ejemplos: 3.2+6j, 0.1+4.55J
Booleanos	bool	Presentan dos valores: *True* y *False* equivalentes a los valores verdadero o falso de la lógica.
Secuencias	str	Cadenas de texto o secuencias de caracteres. Ejemplo: "Buenos días"
	list	Listas de elementos de cualquier tipo que pueden ser modificables. Ejemplos: [1, "ya", True], ['lunes', 'jueves', 'viernes, 'domingo']
	tuple	Las tuplas son listas que no se pueden modificar. Se crean con unos valores y no pueden

		cambiarse. Ejemplos: (1, "ya", True), ('lunes', 'jueves', 'viernes, 'domingo')
	bytes	Secuencias de enteros que representa códigos ASCII y que no se pueden modificar. Ejemplos: b'Python is interesting.', b'\x00\x00\x00\x00\x00'.
	bytearray	Similar a *bytes* pero sí se pueden modificar.
Conjuntos	set	Conjuntos de datos desordenados pero sin repeticiones de elementos. Se pueden modificar. Ejemplos: {3, 5, 1, 7, 4, 10}, {'lunes', 'jueves'}
	frozenset	Similar a *set* pero no se puede modificar.
Diccionarios	dict	Permiten almacenar valores a los que se asigna una clave única para acceder. Ejemplos: {'one': 1, 'two': 2, 'three': 3}

Los operadores son símbolos que nos permiten realizar operaciones con objetos de un programa. Dependiendo de la clase a la que pertenezcan dichos objetos, podremos utilizar unos operadores u otros.

Tipos de operadores	**Operadores disponibles**
Aritméticos	+, -, *, /, // (división entera), % (módulo de la división entera), ** (potencia)
Comparación	==, !=, >, >=, <, >=
Asignación	=, +=, -=, *=, /=, //=, %=, **=
Booleanos o lógicos	and, or, not
Identidad	is, is not

5. Funciones integradas

El intérprete de *Python* dispone de **funciones integradas** o predefinidas que podemos utilizar en cualquiera de nuestros programas y sin necesidad de importar ningún módulo extra.

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Figura 1: Font: https://docs.python.org/3/library/functions.html

5.1. Funciones de entrada y salida

Para solicitar datos por teclado al usuario podemos utilizar la función **input()**. Cuando se ejecuta esta función, el programa queda esperando que el usuario introduzca una cadena de texto por teclado y pulse la tecla *Intro/Enter*.

```
In [7]: texto = input("Escribe una frase, por favor: ")
Escribe una frase, por favor: Buenos días.

In [8]: texto
Out[8]: 'Buenos días.'
```

Figura 2: Uso de input() con enunciado

Si necesitamos pedir por teclado un valor distinto de texto deberemos hacer la conversión al tipo de datos correspondiente. Por ejemplo: **int()** o **float()**.

```
In [14]: entero = int(input("Dame un número entero: ")) ; entero
Dame un número entero: 15
Out[14]: 15

In [15]: real = float(input("Dame un número real: ")) ; real
Dame un número real: 35.298
Out[15]: 35.298
```

Figura 3: Petición de números por teclado

Por su lado, la función **print()** permite mostrar información por pantalla al usuario. Podemos imprimir cadenas de texto, números, valores lógicos, listas, diccionarios, etc.

Existen distintas formas de utilizar esta función. Aquí se pueden ver algunos ejemplos:

```
In [35]: print("Hola %s." % ("Pepito"))
Hola Pepito.

In [36]: print("%s, trae %d tebeos de %.2f€ cada uno." % ("Pepito", 5, 12.5))
Pepito, trae 5 tebeos de 12.50€ cada uno.
```

Figura 4: Uso básico del operador %

```
In [19]: print("{} trae {} tebeos de {:.2f}€ cada uno.".format("Pepito", 5, 12.5))
Pepito trae 5 tebeos de 12.50€ cada uno.
```

Figura 5: Uso básico de {}

```
>>> nom = "Pepito"
>>> num = 5
>>> preu = 12.5
>>> print(f'{nom}, trae {num} tebeos a {preu}€ cada uno.')
Pepito, trae 5 tebeos a 12.5€ cada uno.
```

Figura 6: Uso básico de f-cadenas

6. Módulos en Python

Cuando queremos crear un programa en *Python*, tenemos que crear un fichero de texto plano con extensión *.py*. Cada uno de estos ficheros es considerado un **módulo** en Python.

Para importar un módulo debemos utilizar la palabra reservada **import** seguida del nombre del módulo y de la jerarquía de paquetes a la que pertenezca:

```
import nombre_modulo

import paquete.nombre_modulo
```

import *paquete.subpaquete.nombre_modulo*

En algunas ocasiones nos puede interesar importar sólo algunos elementos de un módulo y no el módulo completo. En ese caso, debemos indicar el elemento a importar o una lista de elementos separados por coma, tal y como se muestra a continuación:

from *nombre_modulo* import *elemento*

from *nombre_modulo* import *elemento*$_1$, *elemento*$_2$, ..., *elemento*$_n$

A partir de entonces, podemos utilizar al nombre del elemento directamente en el programa sin necesidad de anteponer el *namespace* correspondiente.

```
In [23]: from paquete1.modulo11 import resta

In [24]: resta(10, 3)
Out[24]: 7
```

Figura 7: Uso de "from" en un "import"

7. Estructuras de control

En este apartado repasaremos la sintaxis de las distintas estructuras de control que ofrece Python.

7.1. Instrucciones alternativas: if, else, elif

if *condición*:
    instrucción$_{1.1}$
    ...
    instrucción$_{1.n}$
else:
    instrucción$_{2.1}$
    ...
    instrucción$_{2.m}$

```
26  seguir = input("¿Quieres seguir (S/N)?")
27  if seguir == 'S':
28      print("Has elegido seguir.")
29  else:
30      print("Has elegido no seguir.")
31
32
33  num = int(input("Dame un número (1-10):"))
34  if num < 5:
35      print("Sigue intentándolo.")
36  else:
37      print("¡Enhorabuena!")
```

Figura 8: Ejemplo usos básicos de if-else

if *condición*$_1$:
    instrucción$_{1.1}$
    ...
    instrucción$_{1.n}$
[elif condición$_2$:
    instrucción$_{2.1}$
    ...
    instrucción$_{2.m}$
...
elif condición$_p$:
    instrucción$_{p.1}$
    ...
    instrucción$_{p.q}$]
[else:
    instrucción$_{r.1}$
    ...
    instrucción$_{r.s}$]

```
76  num = int(input("Dame un número (1-10):"))
77  if num <= 2:
78      print("Muy deficiente")
79  elif num >= 3 and num <= 4:
80      print("Insuficiente")
81  elif num == 5:
82      print("Aprobado")
83  elif num == 6:
84      print("Bien")
85  elif num >= 7 and num <= 8:
86      print("Notable")
87  elif num >= 9:
88      print("Sobresaliente")
```

Figura 9: Uso de cláusulas elif

7.2. Instrucciones repetitivas: while, for

while *condición*:
    instrucción$_{1.1}$
    ...
    instrucción$_{1.n}$

```
91  opcion = input("¿Quieres seguir? (S/N) ")
92  while ((opcion.upper() != 'S') and (opcion.upper() != 'N')):
93      opcion = input("¿Quieres seguir? (S/N) ")
94  print("Opción correcta: ", opcion)
```

Figura 10: Uso de "while" para validar un dato introducido por teclado

for *elemento* in *secuencia*:
    instrucción$_{1.1}$
    ...
    instrucción$_{1.n}$

```
110 palabra = "HOLA"
111 for i in range(len(palabra)):
112     print(i, '-->', palabra[i])
```

Figura 11: Iterando sobre una secuencia

7.3. Otras instrucciones: break, continue, pass

Python dispone de dos sentencias que permiten alterar el funcionamiento normal de un bucle: *break* y *continue*.

Cuando se utiliza la sentencia **break** dentro de un bucle provoca la finalización del mismo sin hacer caso de la condición del **while** o del iterador del **for**.

Por su parte, la sentencia **continue** da por finaliza la iteración actual y pasa a la siguiente aunque no provoca el fin del bucle completo.

Por último, la sentencia **pass** es una sentencia vacía, es decir, no hace nada. Suele utilizarse para que el programa no dé error de sintaxis cuando se está estructurando un programa pero aún no se ha implementado el conjunto de instrucciones que son necesarias.

```
122 while True:
123     pass
124
125 for i in range(len(palabra)):
126     pass
```

Figura 12: Ejemplo uso de "pass"

8. Definición de funciones

Las funciones se definen con la siguiente sintaxis:

```
def nombre_función(param1, ... , paramn):
        # instrucciones del cuerpo de la función
        return (elem1, elem2, ... , elemn)  #devuelve una tupla
```

La *llamada a una función* dependerá del número de parámetros de entrada que tenga y los valores que devuelva.

Ejemplo: La función *calculadora* recibe dos parámetros de entrada y devuelve cuatro resultados de operaciones.

```python
def calculadora(a,b):
    '''Realiza operaciones básicas.'''
    suma = a + b
    resta = a - b
    producto = a * b
    division = a / b

    return (suma, resta, producto, division)

a = int(input("Dame el operando 1:"))
b = int(input("Dame el operando b:"))
resultado = calculadora(a, b)

for i in range(len(resultado)):
    print("Resultado", i, "-->", resultado[i])
```

Figura 13: Return de varios valores

```
Dame el operando 1:10
Dame el operando b:4
Resultado 0 --> 14
Resultado 1 --> 6
Resultado 2 --> 40
Resultado 3 --> 2.5
```

Figura 14: Ejecución del programa

9. Tipos de datos complejos

La diferencia fundamental entre los *tipos de datos básicos* y los *complejos* es que los primeros son inmutables mientras que los segundos no lo son.

La **mutabilidad** o **inmutabilidad** de un objeto hace referencia a la capacidad de poder modificar su contenido o no, una vez que ha sido creado.

9.1. Tipo de datos LISTA

Las *listas* son un tipo de datos secuencia mutable que permite almacenar elementos ordenados de cualquier tipo.

Para crear una lista, podemos hacerlo de dos formas:

a) Encerrando entre corchetes una lista de elementos separados por comas.

```
In [8]: numeros = [1, 2, 3, 4, 5]; numeros
Out[8]: [1, 2, 3, 4, 5]

In [9]: vocales = ['a', 'e', 'i', 'o', 'u']; vocales
Out[9]: ['a', 'e', 'i', 'o', 'u']
```

Figura 15: Creación de listas con [y]

b) A partir de un tipo secuencia, como una cadena de texto u otra lista, mediante el uso del constructor **list()**.

```
In [26]: numeros
Out[26]: [1, 2, 3, 4, 5]

In [27]: otro = list(numeros); otro
Out[27]: [1, 2, 3, 4, 5]

In [28]: vocales = list("aeiou"); vocales
Out[28]: ['a', 'e', 'i', 'o', 'u']
```

Figura 16: Uso del constructor list()

En la siguiente tabla se recogen algunas operaciones con listas:

Operaciones	Descripción

Acceso a elementos	Utilizamos el operador **[ ]**:
	`In [40]: colores[0] = "negro"`
	
	`In [41]: colores`
	`Out[41]: ['negro', 'verde', 'azul', 'rojo', 'azul']`

Recorrido de listas	Accediendo a los elementos directamente:
	`In [39]: colores`
	`Out[39]: ['rojo', 'amarillo', 'azul', 'verde']`
	
	`In [40]: for color in colores:`
	`   ....:     print(color, end=" ")`
	`   ....:`
	`rojo amarillo azul verde`
	
	Utilizando un índice:
	`In [43]: numeros`
	`Out[43]: [5, 2, 3, 4, 1]`
	
	`In [44]: for i in range(len(numeros)):`
	`   ....:     numeros[i] *= 10`
	`   ....:`
	
	`In [45]: numeros`
	`Out[45]: [50, 20, 30, 40, 10]`

Longitud	Utilizaremos la función **len()**:
	`In [5]: numeros = [5, 2, 3, 4, 1]`
	
	`In [6]: len(numeros)`
	`Out[6]: 5`
	
	`In [7]: colores = ["rojo", "amarillo", "azul", "verde"]`
	
	`In [8]: len(colores)`
	`Out[8]: 4`

Sublistas	Sintaxis: ***nombre_lista[inicio : fin+1: salto]***
Añadir elementos	Utilizando la función **append()**:

	In [70]: numeros = [5, 2, 3, 4, 1] In [71]: numeros.append(8); numeros Out[71]: [5, 2, 3, 4, 1, 8] In [72]: numeros.append([7, 9]); numeros Out[72]: [5, 2, 3, 4, 1, 8, [7, 9]] Concatenando listas con **+**: In [65]: numeros = [5, 2, 3, 4, 1] In [66]: numeros = numeros + [8]; numeros Out[66]: [5, 2, 3, 4, 1, 8] In [67]: numeros = [6, 7] + numeros; numeros Out[67]: [6, 7, 5, 2, 3, 4, 1, 8]
Borrar elementos	Usando el método **pop()** que borra el elemento ubicado en una posición o el último, por defecto: In [77]: numeros = [5, 2, 3, 4, 1] In [78]: numeros.pop() Out[78]: 1 In [79]: numeros.pop(0) Out[79]: 5 In [80]: numeros Out[80]: [2, 3, 4]

9.2. Tipo de datos DICCIONARIO

Un **diccionario** es un tipo de dato compuesto por pares **clave : valor** donde *clave* puede ser cualquier tipo inmutable, habitualmente cadenas de texto o números. En cuanto a los valores, pueden ser de cualquier tipo de datos.

Una característica que los diferencia de las *listas* es que, en este caso, sus elementos no guardan ningún orden.

Podemos crear un diccionario de varias formas:

a) Utilizando el operador { }:

```
In [33]: dic = {'nombre':"Mafalda", 'ama':"Los Beatles", 'odia':"sopa"}

In [34]: dic
Out[34]: {'ama': 'Los Beatles', 'nombre': 'Mafalda', 'odia': 'sopa'}
```

Figura 17: Crear un diccionario usando { }

b) Utilizando el constructor **dict()** y pasándole los pares *clave:valor* correspondientes:

```
In [19]: dic = dict(nombre = "Mafalda", ama = "Los Beatles", odia = "sopa")

In [20]: dic
Out[20]: {'ama': 'Los Beatles', 'nombre': 'Mafalda', 'odia': 'sopa'}
```

Figura 18: Uso de dict() con asignaciones clave=valor

```
In [22]: dic = dict([('nombre', 'Mafalda'), ('ama', 'Los Beatles'), ('odia', 'sopa')])

In [23]: dic
Out[23]: {'ama': 'Los Beatles', 'nombre': 'Mafalda', 'odia': 'sopa'}
```

Figura 19: Uso de dict() con secuencias de (clave, valor)

c) Creando un diccionario vacío y, después, utilizando el operador de asignación para cada uno de los pares *clave:valor* que queremos añadir:

```
In [12]: dic = {}

In [13]: dic["nombre"] = "Mafalda"

In [14]: dic["ama"] = "Los Beatles"

In [15]: dic["odia"] = "sopa"

In [16]: dic
Out[16]: {'ama': 'Los Beatles', 'nombre': 'Mafalda', 'odia': 'sopa'}
```

Figura 20: Creación de un diccionario

En la siguiente tabla se recogen algunas operaciones con diccionarios:

Operaciones	Descripción
Acceso a elementos	Sintaxis: ***nombre_diccionario[clave]***

Longitud	Utilizaremos la función **len()**:
	`In [113]: dic1 = {'nombre':"Mafalda", 'ama':"Los Beatles", 'odia':"sopa"}`
	`In [114]: len(dic1)`
	`Out[114]: 3`

Recuperar elementos	Para recuperar la lista de elementos usaremos **items()**:
	`In [91]: dic1 = {'nombre':"Mafalda", 'ama':"Los Beatles", 'odia':"sopa"}`
	`In [92]: dic1.items()`
	`Out[92]: dict_items([('ama', 'Los Beatles'), ('nombre', 'Mafalda'), ('odia', 'sopa')])`
	Para recuperar la lista de claves usaremos **keys()**:
	`In [34]: dic1 = {'nombre':"Mafalda", 'ama':"Los Beatles", 'odia':"sopa"}`
	`In [35]: dic1.keys()`
	`Out[35]: dict_keys(['ama', 'nombre', 'odia'])`
	Para recuperar la lista de valores usaremos **values()**:

	`In [1]: dic1 = {'nombre':"Mafalda", 'ama':"Los Beatles", 'odia':"sopa"}`
	
	`In [2]: dic1.values()`
	`Out[2]: dict_values(['Los Beatles', 'Mafalda', 'sopa'])`

Recorrer diccionario	Utilizando el método **keys()** para recorrer solamente las *claves*:
	
	`In [82]: dic1 = {'nombre':"Mafalda", 'ama':"Los Beatles", 'odia':"sopa"}`
	
	`In [83]: for k in dic1.keys():`
	`....:  print(k, end=" ")`
	`....:`
	`ama nombre odia`
	
	Utilizando el método **items()** para recorrer tanto las *claves* como los *valores*:
	
	`In [103]: dic1 = {'nombre':"Mafalda", 'ama':"Los Beatles", 'odia':"sopa"}`
	
	`In [104]: for k, v in dic1.items():`
	`.....:  print(k,'-->',v)`
	`.....:`
	`ama --> Los Beatles`
	`nombre --> Mafalda`
	`odia --> sopa`

Borrar elementos	El método **pop()** borra el elemento correspondiente a la *clave* que se pasa como argumento. Sintaxis **pop(*clave*[,*valor_defecto*]]**

10. Excepciones

Una **excepción** se produce cuando ocurre un error de ejecución en un programa. En ese momento, el programa acaba y Python muestra un mensaje de error.

Podemos manejar las excepciones que se producen en la ejecución de un programa y dar una respuesta personalizada. Para ello debemos utilizar el bloque **try...except**.

Su sintaxis es la siguiente:

```
try:
        instrucciones que pueden provocar una excepción
except nombre_excepción1:
        instrucciones para el tipo de excepción 1
except:
        instrucciones para excepciones generales
else:
        instrucciones que se ejecutarán cuando no se produzca una excepción
```

finally:
instrucciones que se ejecutarán al final del bloque

```
try:
    dato = int(input("Introduce un entero: "))
    dato = 100 / dato
except ValueError:
    print("Error. El dato no es un entero.")
except ZeroDivisionError:
    print("Error. El dato debe ser distinto de cero.")
except:
    print("\nError genérico.")
else:
    print("El resultado es:", dato)
finally:
    print("Fin del programa.")
```

Figura 21: Especificamos varios tipos de excepciones y el general

11. Fuentes de información

- Página oficial del lenguaje Python: https://www.python.org/
- Qué es Python: https://es.wikipedia.org/wiki/Python
- Página oficial del IDE Thonny: https://thonny.org/