

GRADO EN INGENIERÍA INFORMÁTICA

Módulo Fundamentos de Informática

METODOLOGÍA DE PROGRAMACIÓN

D. Roger Clotet Martínez



viu

**Universidad
Internacional
de Valencia**



Este material es de uso exclusivo para los alumnos de la VIU. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la VIU, sin autorización expresa de la misma.

Edita

Universidad Internacional de Valencia

Grado en
Ingeniería Informática

Metodología de Programación

Módulo Fundamentos de Informática

6ECTS

D. Roger Clotet Martínez

Índice

TEMA 1. TIPOS DE DATOS DEL LENGUAJE DE ALTO NIVEL Y SU REPRESENTACIÓN INTERNA.....	9
1.1. Entero (int)	9
1.2. Real (float)	10
1.3. Booleano (bool)	12
1.4. Caracteres (str)	12
 TEMA 2. REFERENCIAS DE MEMORIA Y MEMORIA DINÁMICA	19
2.1. Memoria dinámica.....	20
2.2. Administración de memoria	20
2.2.1. Ejemplo C	21
2.3. Apuntadores	22
2.3.1. Ventajas vs Desventajas de los Apuntadores	23
2.4. Python	23
2.4.1. Garbage Collector.....	25
 TEMA 3. ENCAPSULAMIENTO Y OCULTAMIENTO DE LA INFORMACIÓN.....	27
3.1. Encapsulamiento	28
3.2. Clases en programación OO	28
3.2.1. Clases en programación OO	30
3.2.2. Conceptos OO	30
 TEMA 4. DISEÑO MODULAR Y CREACIÓN DE BIBLIOTECAS.....	33
4.2. Diseño modular	34
4.3. Diseño modular en Python	35
4.3.1. Packets.....	36
 TEMA 5. HERRAMIENTAS DE DEPURACIÓN, PRUEBAS Y VALIDACIÓN.....	41
5.1. Herramientas de depuración.....	41
5.1.1. print().....	42
5.1.2. Spyder Debugger.....	43
5.2. Pruebas y Validación.....	46

5.2.1. Caja Negra	48
5.2.2. Caja Blanca.....	50
5.2.3. Utilidad.....	51
5.2.4. ¿Cómo?	52
5.2.5. Evaluar resultados	53
 TEMA 6. GESTIÓN DE ERRORES	 55
6.1. Control de parámetros	56
6.2. Excepciones en el código.....	58
6.3. Gestión de excepciones en Python.....	59
6.3.1. Sin gestión	60
6.3.2. Excepciones Predefinidas	61
6.3.3. Gestión de Excepciones	63
6.3.4. Lanzar una excepción	66
6.3.5. Código que siempre se ejecuta.....	67
 TEMA 7. MANTENIMIENTO DEL SOFTWARE	 69
7.1. Código heredado	71
7.2. Leyes de Lehman	72
7.2.1. Continuidad del Cambio	72
7.2.2. Incremento de la Complejidad.....	73
7.2.3. Evolución del Programa.....	74
7.2.4. Conservación de la Estabilidad Organizacional	74
7.2.5. Conservación de la Familiaridad	74
7.2.6. Crecimiento continuo	74
7.2.7. Declive de la calidad.....	75
7.2.8. Retroalimentación del sistema.....	75
7.3. Problemas generales.....	75
7.4. En el Grado Ingeniería Informática	75
 TEMA 8. E/S (INPUT/OUTPUT), FICHEROS	 77
8.1. Entra/Salida.....	78
8.2. Ficheros en Python.....	78

8.2.1. Open()	79
8.2.2. read()	81
8.2.3. readline()	83
8.2.4. readlines() / list()	84
8.2.5. write()	85
8.2.6. tell() / seek()	86
8.2.7. close()	88
 TEMA 9. PROYECTO INFORMÁTICO DE PROGRAMACIÓN	89
9.1. Ejemplo de un proyecto	90
9.1.1. Componente 1: Gasolinera	91
9.1.2. Componente 2: Refinería	93
9.1.3. Componente 3: Integración componentes	96
 GLOSARIO	99
 ENLACES DE INTERÉS	103
 BIBLIOGRAFÍA	105
Referencias bibliográficas	105
Bibliografía recomendada	105

Leyenda



Glosario

Términos cuya definición correspondiente está en el apartado "Glosario".

Tema 1.

Tipos de datos del lenguaje de alto nivel y su representación interna

Los **lenguajes de programación de alto nivel** permiten **abstraerse** de cómo se representan internamente los datos, cómo se manipulan y cómo se implementa la lógica de los algoritmos, permitiendo al programador centrarse en la tarea de resolver el problema. Posteriormente, se realizará una transformación del algoritmo de alto nivel a un conjunto de instrucciones, lógica y datos que permita su ejecución en el sistema.

En este primer capítulo veremos cómo son **representados internamente** los datos básicos, en nuestro caso particular en cómo se representan en el lenguaje **Python** y cuáles son sus limitaciones. En la teoría, por ejemplo, podemos tener infinitos números enteros, pero físicamente solo seremos capaces de representar un número finito según las **limitaciones** del **hardware y/o software** utilizado.

1.1. Entero (int)

A diferencia de otros lenguajes donde tenemos un tamaño máximo para la representación de los **enteros** (*int*), en **Python** los datos de tipo entero **se almacenan con la precisión necesaria** para representar el número **siempre que** el sistema tenga suficiente **memoria disponible**.

Ejemplos:

```
In [5]: bin(2)
Out[5]: '0b10'
```

Figura 1. Ejemplo representación binaria del entero 2. Fuente: Elaboración propia.

[illegible]

Figura 2. Ejemplo representación binaria del entero -2^{64} . Fuente: Elaboración propia.

[illegible]

Figura 3. Ejemplo representación binaria del entero 2^{1000} . Fuente: Elaboración propia.

Como podemos ver en los tres ejemplos (ver figura 1, figura 2 y figura 3), la representación binaria de un entero se adapta al tamaño de este en el caso de Python. En **otros lenguajes** de programación, el tamaño ocupado por un entero es fijo, por ejemplo 32 bits, y los valores máximo y mínimo que se pueden representar están **limitados** a los que se puedan representar con la cantidad de **bits asignados**.

Si queremos saber la longitud en bits de la representación de cualquier entero, podemos usar la función *bit_length()* (ver figura 4).

```
In [11]: n = 128
In [12]: n.bit_length()
Out[12]: 8
In [13]: n = -2**64
In [14]: n.bit_length()
Out[14]: 65
```

Figura 4. Ejemplo de uso de `bit_length()`. Fuente: Elaboración propia.

1.2. Real (float)

Python implementa los **reales** (*float*) a bajo nivel mediante **64 bits**, en otros lenguajes existen los *float* de 32 bits y los *doble* de 64 bits. Con esos 64 bits y utilizando el **estándar IEEE 754 (1 bit para el signo, 11 para el exponente, y 52 para la mantisa)** se pueden representar valores que van desde $\pm 2,2250738585072020 \times 10^{-308}$ hasta $\pm 1,7976931348623157 \times 10^{308}$.

Signo	Exponente	Mantisa
1 bit	11 bits	52 bits

$$\text{Valor} = (-1)^{\text{Signo}} * 1.\text{Mantisa} * 2^{(\text{Exponente} - 1023)}$$

Tabla 1. Cómo se calcula el valor con estándar IEEE 754. Fuente: Pedro Gomis, material VIU 04GIIN (Gomis, P., 2017).

La norma IEEE 754 actualizada en 2008 incorpora el formato decimal64 que utiliza la base decimal **para mejorar los errores de representación binaria** (Zuras, 2008). **Python** incorpora la función **Decimal** del módulo decimal con este fin.

Existen ejemplos de errores de precisión con reales (*float*). En la figura 5 podemos observar como una simple suma de dos números reales con solo un decimal no es realizada con la precisión necesaria y nos regresa un resultado erróneo ($1,1 + 2,2 = 0,3$ y no a $3,3000000000000003$). Igualmente, como podemos ver en la figura 6, en el caso de una resta nos encontramos con un error de precisión:

```
In [1]: 1.1 + 2.2
Out[1]: 3.3000000000000003
```

Figura 5. Ejemplo de error de precisión en la suma de dos reales. Fuente: Elaboración propia.

```
In [2]: 0.1 + 0.1 + 0.1 - 0.3
Out[2]: 5.551115123125783e-17
```

Figura 6. Ejemplo de error de precisión en la suma y resta de cuatro reales. Fuente: Elaboración propia.

Esto pueden inducir que nuestros programas realicen cálculos incorrectos que, si bien los errores son pequeños, pueden significar errores graves ya que son operaciones que requieren gran precisión u operaciones que se realicen millones de veces.

Usando el modulo Decimal, el cual debemos importar dentro de nuestro código para poder utilizarlo usando lo siguiente::

```
from decimal import *
```

La precisión aumenta y algunos errores de precisión ya no se producen. Por ejemplo $0,1 + 0,2$ ya da el valor correcto (ver figura 7), pero en otros casos como $0,1 + 0,1 + 0,1$ aún sigue faltando precisión (ver figura 7):

```
In [6]: from decimal import *
In [7]: Decimal(0.1) + Decimal(0.2)
Out[7]: Decimal('0.300000')

In [8]: Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)
Out[8]: Decimal('1.11022E-17')
```

Figura 7. Dos ejemplos del uso de Decimal, uno corrige la falta de precisión y en el otro aún no es suficiente. Fuente: Elaboración propia.

Tenemos también otra opción que nos permite Python que es indicar la precisión deseada usando:

`getcontext().prec`

```
In [12]: getcontext().prec = 1000  
In [13]: Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)  
Out[13]: Decimal('2.77555756156289135105907917022705078125E-17')
```

pero aun así, en algunos casos no es suficiente. Por ejemplo, (ver figura 8) en el caso anterior de sumar $0,1 + 0,1 + 0,1 - 0,3$ aún no regresa el resultado correcto de 0:

Figura 8. Ejemplos de uso de una precisión de 1000, pero que resulta insuficiente para obtener el resultado sin aproximaciones.
Fuente: Elaboración propia.

1.3. Booleano (bool)

Aun cuando podemos realizar operaciones booleanas sobre elementos que no son propiamente booleanos, que son definidos como variables con los valores **Verdadero** o **Falso**. En este caso podemos obtener por lo general que la variable será evaluada como Verdadera para cualquier valor diferente de 0 o nulo (elementos vacíos, como por ejemplo un vector) y Falso en caso contrario. Pero no es recomendable usar de esta forma los operaciones con booleanos.

Si usamos variables booleanas estrictas (Verdadero o Falso), son representadas internamente como **0 para Falso** y **1 para Verdadero** (ver figura 9).

```
In [19]: v = True  
In [20]: f = False  
In [21]: bin(v)  
Out[21]: '0b1'  
In [22]: bin(f)  
Out[22]: '0b0'
```

Figura 9. Ejemplo de representación de Falso y Verdadero, el 0b significa que es representación binaria.
Fuente: Elaboración propia.

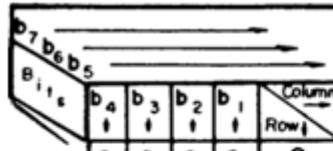
1.4. Caracteres (str)

En el año 1963 se estableció el estándar ASCII para homogenizar la representación de los caracteres en diferentes sistemas informáticos. El mismo estaba basado en los códigos usados para la transmisión por telégrafo de mensajes de texto e, inicialmente, solo era posible representar un número limitado de caracteres basados en los usados en el alfabeto latino en inglés:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN OPQRSTUVWXYZ[\  
]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Para cada uno de ellos, se estableció una codificación binaria y también se incluyeron algunos códigos de control para controlar los dispositivos de impresión, por ejemplo, hay un código para indicar el inicio del bloque de texto (STX). Es lo que se conoce como tabla ASCII (ver figura 10).

USASCII code chart



Column	0	1	2	3	4	5	6	7
Row 0	NUL	DLE	SP	@	P	`	p	
Row 1	SOH	DC1	!	1	A	Q	o	q
Row 2	STX	DC2	"	2	B	R	b	r
Row 3	ETX	DC3	#	3	C	S	c	s
Row 4	EOT	DC4	\$	4	D	T	d	t
Row 5	ENQ	NAK	%	5	E	U	e	u
Row 6	ACK	SYN	&	6	F	V	f	v
Row 7	BEL	ETB	'	7	G	W	g	w
Row 8	BS	CAN	(8	H	X	h	x
Row 9	HT	EM)	9	I	Y	i	y
Row 10	LF	SUB	*	:	J	Z	j	z
Row 11	VT	ESC	+	;	K	[k	{
Row 12	FF	FS	,	<	L	\	l	
Row 13	CR	GS	-	=	M]	m	}
Row 14	SO	RS	.	>	N	^	n	~
Row 15	SI	US	/	?	O	_	o	DEL

Figura 10. La carta de Código ASCII de 1968 de los EE.UU.

Fuente: Empleado desconocido del Gobierno de Estados Unidos (dominio público).

Inicialmente, se usaron 7 bits (ver figuras 11 y 12) y, después, fue ampliada hasta 8 bits (ver figura 13) donde se introdujeron, entre otros, caracteres como las vocales acentuadas o la ñ.

Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]
1	1	[STAR OF HADING]	33	21	!
2	2	[STAR OF TEXT]	34	22	"
3	3	[END OF TEXT]	35	23	#
4	4	[END OF TRANSMISSION]	36	24	\$
5	5	[ENQUIRY]	37	25	%
6	6	[ACKNOWLEDGE]	38	26	&
7	7	[BELL]	39	27	'
8	8	[BACKSPACE]	40	28	(
9	9	[HORIZONTAL TAB]	41	29)
10	A	[LINE FEED]	42	2A	*
11	B	[VERTICAL TAB]	43	2B	+
12	C	[FORM FEED]	44	2C	`
13	D	[CARRIAGE RETURN]	45	2D	-

Decimal	Hex	Char	Decimal	Hex	Char
14	E	[SHIFT OUT]	46	2E	.
15	F	[SHIFT IN]	47	2F	/
16	10	[DATA LINK ESCAPE]	48	30	0
17	11	[DEVICE CONTROL 1]	49	31	1
18	12	[DEVICE CONTROL 2]	50	32	2
19	13	[DEVICE CONTROL 3]	51	33	3
20	14	[DEVICE CONTROL 4]	52	34	4
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5
22	16	[SYNCHRONOUS IDLE]	54	36	6
23	17	[ENG OF TRANS BLOCK]	55	37	7
24	18	[CARCEL]	56	38	8
25	19	[END OF MEDIUM]	57	39	9
26	1A	[SUBSTITUTE]	58	3A	:
27	1B	[ESCAPE]	59	3B	;
28	1C	[FILE SEPARATOR]	60	3C	<
29	1D	[GROUP SEPARATOR]	61	3D	=
30	1E	[RECORD SEPARATOR]	62	3E	>
31	1F	[UNIT SEPARATOR]	63	3F	?

Figura 11. Tabla ASCII 7 bits del código 0 al 63. Fuente: Pedro Gomis, material VIU 04GIIN (Gomis, P., 2017).

Decimal	Hex	Char	Decimal	Hex	Char
64	40	@	96	60	`
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r

Decimal	Hex	Char	Decimal	Hex	Char
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[123	7B	{
92	5C	\	124	7C	
93	5D]	125	7D	}
94	5E	^	126	7E	-
95	5F	_	127	7F	[DEL]

Figura 12. Tabla ASCII 7 bits del código 64 al 127. Fuente: Pedro Gomis, material VIU 04GIIN (Gomis, P., 2017).

La tabla de 8 bits contiene los mismos caracteres que la de 7 (ver figuras 11 y 12) y añade los siguientes (ver figura 13):

Decimal	Carácter	Decimal	Carácter	Decimal	Carácter	Decimal	Carácter
128	Ç	160	á	192	Ł	224	Ó
129	ü	161	í	193	ł	225	õ
130	é	162	ó	194	Ť	226	Ô
131	â	163	ú	195	ŧ	227	Ò
132	ă	164	ñ	196	—	228	ö
133	à	165	Ñ	197	†	229	Õ
134	å	166	ª	198	ä	230	μ
135	ç	167	º	199	Â	231	þ
136	ë	168	¿	200	Ľ	232	ƒ
137	ě	169	•	201	Ŧ	233	Ú
138	è	170	¬	202	Ľ	234	Û
139	ï	171	½	203	Ŧ	235	Ù
140	î	172	¼	204	Ŧ	236	Ý
141	ì	173	¡	205	=	237	Ý
142	Ä	174	«	206	Ŧ	238	-
143	Å	175	»	207	α	239	'
144	É	176	☐	208	ö	240	-
145	æ	177	■	209	Ð	241	±
146	Æ	178	■	210	Ê	242	=
147	ô	179		211	Ë	243	¾
148	ö	180	†	212	È	244	¶

Decimal	Carácter	Decimal	Carácter	Decimal	Carácter	Decimal	Carácter
149	ò	181	Á	213	l	245	§
150	ü	182	Â	214	í	246	÷
151	ù	183	À	215	î	247	¸
152	ÿ	184	©	216	ï	248	°
153	Ö	185	¶	217		249	ˆ
154	Ü	186	¶	218		250	˙
155	ø	187	¶	219		251	¹
156	£	188	¶	220		252	³
157	Ø	189	¢	221		253	²
158	x	190	¥	222		254	
159	f	191		223		255	

Figura 13. Tabla ASCII 8 bits del código 128 al 255. Fuente: Pedro Gomis, material VIU 04GIIN (Gomis, P., 2017).

Aun cuando podemos representar los **caracteres de forma individual** y en muchos lenguajes existe el tipo de dato para una sola letra (por ejemplo, tipo *char* en C), en el caso de **Python no** tenemos la opción de un único carácter, siempre debemos trabajar con el tipo *str*, o *string* que es un **lista de caracteres** (ver figura 14),

```
In [25]: x = 'a'
In [26]: type(x)
Out[26]: str
```

Figura 14. Un solo carácter pero también es del tipo *str* en Python. Fuente: elaboración propia.

Adicionalmente, el tipo *string* en Python tiene asociadas algunas constantes y funciones que son útiles para trabajar con listas de caracteres. Algunas funciones interesantes:

- *str.capitalize()*

Regresa la lista de caracteres inicial pero poniendo la primera letra en mayúscula (ver figura 15).

```
In [17]: s = 'hola mundo !'
In [18]: s.capitalize()
Out[18]: 'Hola mundo !'

In [19]: s.isalpha()
Out[19]: False

In [20]: s.isdigit()
Out[20]: False
```

Figura 15. Introducimos una cadena en minúsculas y ponemos la primera en mayúsculas al aplicar la función *capitalize()*. Fuente: Elaboración propia.

- *str.isalpha()*

Regreso Verdadero si la lista de caracteres es no vacía y todos los caracteres son letras. En la figura 15 podemos ver que no regresa Falso, pues tenemos '!' que no es considerado una letra.

- *str.isdigit()*

Regreso Verdadero si la lista de caracteres es no vacía y todos los caracteres son números. En la figura 15 podemos ver que una cadena no está compuesta por puros números.

Hay muchas más, pero no es el objetivo de este manual enumerarlas todas, para el listado completo pueden consultar la documentación de Python en <https://docs.python.org/3.6/library/stdtypes.html#string-methods>

Tema 2.

Referencias de memoria y memoria dinámica

La **memoria** es un **recurso "escaso"** (cada vez menos, pero también cada vez los programas necesitan más memoria) en los ordenadores. Podemos encontrar memoria integrada en la misma **CPU** (memoria caché), la memoria propiamente dicha (**RAM**) (ver figura 16) y la memoria virtual que se usa para simular una ampliación de la memoria RAM cuando esta se agota usando espacio en el disco (ver figura 17).



Figura 16. RAM. Fuente: Pixabay (CC0 Creative Commons).



Figura 17. Disco Fuente: Pixabay (CC0 Creative Commons).

2.1. Memoria dinámica

La memoria dinámica se refiere a aquella **memoria** que **no** puede ser **definida en tiempo de programación**, ya que no se conoce o no se tiene idea del número de las variables a considerarse y/o su tamaño.

La memoria dinámica **permite solicitar memoria en tiempo de ejecución/interpretación**, por lo que cuanto más memoria se necesite, más se solicita al sistema operativo.

Durante la ejecución de un programa, algunas **variables se crean y se destruyen**, por lo tanto, la estructura de **memoria** se va **dimensionando a los requerimientos** del programa, evitándonos así perder datos y optimizar el uso de la memoria.

2.2. Administración de memoria

Podemos imaginarnos la **memoria** del computador como un gran **vector** formado por casillas numeradas (direcciones) (ver tabla 2). Cada casilla tiene una dirección única y su **contenido** puede ser **datos, instrucciones o otra dirección** de programas.

Dirección 1	Dato o Instrucción 1	← Casilla / palabra
Dirección 2	Dato o Instrucción 2	
Dirección n	Dato o Instrucción n	

Tabla 2. Tabla Dirección - Datos o Instrucciones. Fuente: Elaboración Propia

El vector podrá guardar la información en múltiplos del tamaño de las casillas, aun así cuando por ejemplo tengamos el entero 128 que ocupa solo 8 bits, como vimos anteriormente (ver figura 4), en el caso que las casillas sean de 4 bytes (32 bits), desperdiciaremos el espacio.

Si, por ejemplo, la memoria es de 4 Gbytes y cada casilla (palabra) es de 4 bytes, tendríamos 1.073.741.824 palabras de memoria.

4Gbytes = $4 * 1024$ Mbytes = $4 * 1024 * 1024$ Kbytes = 4.294.967.296 bytes de memoria

Cada palabra 4 bytes = 1.073.741.824 palabras

Python no permite gestionar directamente la memoria, por eso veremos unos ejemplos en el lenguaje C, que sí que lo permite.

2.2.1. Ejemplo C

Asignación dinámica de memoria

Una de las características del lenguaje C es que permite que el programador maneje de manera dinámica la memoria para contener datos y estructuras de cualquier tamaño de acuerdo a las necesidades en tiempo de ejecución.

¿Cómo se hace?

Usando llamadas al sistema especiales que nos permiten pedir y liberar memoria. Veremos algunas de ellas para ilustrar cómo funciona.

Función malloc()

*void *malloc(size_t size)*

Regresará un apuntador del tipo *void **, el cual es el inicio en memoria de la porción reservada de tamaño *size*. Si no puede reservar esa cantidad de memoria, la función regresa un apuntador nulo o *NULL*. C asume que el apuntador puede ser convertido a cualquier tipo de dato que necesite el programador, por ejemplo a entero o a real.

El tipo de argumento *size_t* está definido en la cabecera *stddef.h* y es un tipo entero sin signo.

*char *cp; cp = (char *) malloc(100);*

Intenta obtener 100 bytes y asignar a *cp* la dirección de memoria de inicio del bloque en caso de que la petición resulte correcta, en caso de error regresara *NULL*.

Función sizeof()

Devuelve el tamaño en bytes del tipo (también para estructuras de datos más complejas definidas por el usuario).

Ejemplo:

*int *ip;*

*ip = (int *) malloc(100 * sizeof(int));*

El compilador de C requiere hacer una conversión del tipo de puntero a *void* a puntero a entero. La forma de lograr la coerción (*cast*) es usando (*int **) que permite convertir un apuntador a *void* a un apuntador a tipo *int*, respectivamente. Hacer la conversión al tipo de apuntador correcto asegura que la aritmética (por ejemplo cuando hagamos una suma o resta para movernos entre posiciones de un vector de enteros) con el apuntador funcionará de forma correcta.

Es una buena práctica usar *sizeof()*, aun si se conoce el tamaño actual del dato que se requiere, ya que, de esta forma, el código se hace independiente del dispositivo (portabilidad). Algunos tipos de datos pueden cambiar el espacio que ocupan en memoria según la plataforma.

Función *free()*

Cuando se ha terminado de usar una porción de memoria, siempre se deberá liberar usando la función *free()*. Esta función permite que la memoria liberada esté disponible nuevamente, quizás para otra llamada de la función *malloc()*.

La función *free()* toma un apuntador como un argumento y libera la memoria a la cual el apuntador hace referencia.

Ejemplo:

```
free(ip);
```

2.3. Apuntadores

Entre los datos que pueden almacenarse dentro de la memoria están las **direcciones a otros datos**. A estos se les conoce como **apuntadores**.

Cuando se almacenan datos en la memoria, a estos se les asigna un **espacio de memoria** según el tamaño, dependiendo de la definición del tipo dato y en la plataforma que nos encontremos. Teniendo en cuenta que el tamaño siempre será un múltiple del tamaño de la palabra, no podemos direccionar media palabra, así que las direcciones y los datos siempre empezarán al inicio de la dirección indicada.

Ejemplo:

<i>int</i>	<i>X;</i>	<i>X ocupará 2 bytes de memoria</i>
<i>long</i>	<i>Y;</i>	<i>Y ocupará 4 bytes de memoria</i>
<i>char</i>	<i>C;</i>	<i>C ocupará 1 bytes de memoria</i>

En el caso de los apuntadores, generalmente ocupan una casilla o palabra de memoria, y el contenido indica la dirección de otro dato.

Ejemplo C (ver tabla3):

int X;

*int * ApX;* (en C *tipo ** significa declarar una variable del tipo puntero al tipo, en este caso ApX es un puntero a entero).

ApX = & X; (en C *& variable* nos retorna un puntero a la variable, en este caso un puntero a la variable X).

Dirección	Nombre	Contenido
...
1026	x	602
...
1040	ApX	1026
...

Tabla3. Tabla con dirección - nombre - contenido de la memoria. Fuente: Elaboración propia.

2.3.1. Ventajas vs Desventajas de los Apuntadores

Ventajas

- Generar elementos bajo demanda, i.e. asignación dinámica de memoria.
- Manipular y recorrer grandes espacios de memoria.
- Generar estructuras de datos complejas.
- Parámetros de entrada/salida para funciones, i.e. parámetros por referencia.

Desventajas

- Mayor consumo de memoria.
- Programación avanzada, caótica y/o complicada.
- Programación más susceptible a errores difíciles de detectar y corregir.
- Dificultad para leer y comprender código.

2.4. Python

En Python (y en otros lenguajes como Java), no es la declaración de la variable la que debe reservar memoria. Una **variable**, desde el punto de vista del programador, **no ocupa memoria**. Lo que **ocupa memoria** es el **objeto** al que apunta (**su contenido, su valor**).

En Python, las variables no son un lugar de memoria que contienen un valor, sino que se asocian, o refieren, a un lugar de memoria que contiene ese valor.

Python reserva automáticamente espacio para guardar estos valores.

Con la función `id()` podemos saber la dirección en memoria de una variable (ver figuras 18 y 19).

```
In [1]: x = 1
In [2]: id(x)
Out[2]: 1418652112

In [3]: y = x
In [4]: id(y)
Out[4]: 1418652112

In [5]: z = 1
In [6]: id(z)
Out[6]: 1418652112
```

Figura 18. Ejemplos de dirección en memoria en Python para diferentes variables. Fuente: Elaboración propia.

```
In [7]: q = 'hola'
In [8]: id(q)
Out[8]: 1511982409856

In [9]: q = 1
In [10]: id(q)
Out[10]: 1418652112
```

Figura 19. Ejemplos de dirección en memoria en Python para diferentes variables. Fuente: Elaboración propia.

En la figura 18 podemos observar que si `x` y `z` tienen el **mismo valor** (1), la **dirección** en memoria donde está ese valor será **igual para las tres variables** en el ejemplo, la dirección 1418652112. Por el contrario, una misma **variable** cuando **cambia de valor, cambia** también su **dirección**, como se ve en la figura 19, cambiando la dirección de la inicial para `q` de 1511982409856 cuando apunta donde está la cadena de caracteres 'hola' a la dirección 1418652112 que es donde se encuentra el valor 1. Aquí tendríamos que la posición 1418652112 donde está el 1 es apuntado por 4 variables `x` y `z` `q`. El número de variables que apuntan es un dato interesante pues le sirve al sistema para determinar qué posiciones en memoria se pueden liberar, como veremos en el siguiente punto con el *Garbage Collector*.

Aun cuando se suponen que no podemos saber la dirección de un objeto en Python. Usando la función `id(object)` se obtienen la dirección del objeto en memoria, según los detalles de implementación para la versión 3 de Python, eso puede cambiar en un futuro.

`id(object)`

Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

CPython implementation detail: *this is the address of the object in memory.*

<https://docs.python.org/3/library/functions.html#id> (Python Software Foundation, Python 3 documentation, 2018)

2.4.1. Garbage Collector

Como comentábamos en el punto anterior, internamente se guarda un contador con el número de variables que referencian un valor/objeto. Cuando este contador llega a 0, ese espacio es susceptible de ser liberado en la siguiente iteración del **Garbage Collector**.

Internamente, se tienen algoritmos para **optimizar el uso de memoria**, pero no es que automáticamente cuando se llega a 0 apuntadores/referencias se libera el espacio. Se tienen en cuenta varios factores adicionales, por ejemplo, los valores creados más recientemente serán destruidos más tarde que los valores creados hace más tiempo (son menos probables de ser utilizados de nuevo).



Garbage Collector. Fuente: Pixabay, CC0 Creative Commons.

Tema 3.

Encapsulamiento y ocultamiento de la información

Cuando estamos programando, **no siempre es recomendable que todo nuestro código** con sus funciones y variables **sea visible externamente** para otros programas.

A veces, puede ser útil para hacer **independiente la implementación interna**, por ejemplo, cómo se hacen los cálculos o cómo se guardan los datos. Al ofrecer una **interfaz** estable y sin detalles internos, podemos hacer cambios internos (ya sea por correcciones, por mejoras, por cambio de plataforma de base de datos, entre otros) sin que la interoperabilidad se vea afectada. **Por seguridad**, a veces nos es recomendable que sea visible cómo se hacen los cálculos internos o qué variables estamos usando para hacer cálculos intermedios que podrían ayudar a un programador mal intencionado a hacer más fácilmente su trabajo.

También, sirve para **separar** la fase de **diseño** de la de **implementación**. En el diseño podemos hacer el esqueleto de cómo funcionará nuestro programa y qué módulos/funciones necesitaremos sin preocuparnos de la parte interna. En la fase de implementación ya veremos cómo lo hacemos para que las funciones hagan lo esperado de forma correcta e eficiente.

3.1. Encapsulamiento

Consiste en **bloquear el acceso** a determinadas **funciones y/o variables** de una **clase**.

En algunos lenguajes de programación, por ejemplo en Java, esto se consigue indicando *public* o *private* en la declaración de la función o variable.

En **Python**, no podemos indicar en la declaración esta propiedad, el acceso a una variable o función **se determina** por cómo está formado el **nombre** de la misma. Si el nombre **empieza por dos guiones bajos**, en este caso será privada, si no es así, será pública.

__privado()

publico()



Encapsulamiento. Fuente: Pixabay, CC0 Creative Commons.

Es **importante** que en el caso de declarar una función privada, **no** utilicemos **dos guiones bajos para terminar el nombre**. Sino **Python** lo interpretará como una **función espacial**, que son funciones que son llamadas de forma automática en determinadas circunstancias.

En el caso de que el programador conozca el nombre de la variable o la función e intente llamarla o utilizar la variable, obtendrá un error de ejecución como si no existiese, pues el interpretador/compilador no le dará acceso.

3.2. Clases en programación OO

Aun cuando no está entre los temas de esta asignatura, veremos brevemente algunos conceptos de **programación orientada a objetos** (OO) que nos servirán para comprender algunos conceptos. En otras asignaturas del grado será abordado el tema de la OO con más detalle y profundidad.

Las clases son la **piedra angular** de la **programación Orientada a Objetos (OO)**. **Permiten abstraer los datos** (variables) **y** sus **operaciones** asociadas (funciones) al modo de una caja negra. En la figura 20 podemos ver un diagrama de clases, cada clase tiene su nombre que la identifica, por ejemplo 'Noticia'. Dentro de la clase tenemos dos bloques, en la parte superior tenemos las variables públicas de la misma con su respetivo tipo y, en la parte inferior, tenemos las funciones. Por ejemplo, para 'Noticia' tenemos las variables: Usuario, Titulo, Topico, Texto, Texto_Extendido, Categoria y Publicada y para las funciones tenemos: Insertar(), Modificar(), Eliminar() y Consultar().



Figura 20. Ejemplo de un diagrama de clases, indicando variables y operaciones. Las variables son los pares nombre variable - tipo variable, por ejemplo, Usuario: Cadena. Las operaciones se distinguen por (), nombreOperacion(), por ejemplo, Insertar().
 Fuente: The Photographer.

A continuación, de manera muy esquemática se incluye un poco más de información sobre OO.

3.2.1. Clases en programación OO

Las clases nos permiten tener o proporcionar una serie de características:

- **Abstracción**

Caja Negra: lo importante es que hace y no cómo lo hace.

- **Encapsulamiento**

Consiste en bloquear el acceso a determinadas funciones y/o variables de una clase.

- **Jerarquía**

Raíz → Ramas → Hojas

Padre/Madre → Hijo/Hija → Nietos

Al establecer la relación entre las diferentes clases (ver figura 21), permiten reaprovechar el código, pues los hijos heredan las funcionalidades de los padres. Por ejemplo, podemos tener una jerarquía que defina los animales mamíferos, en la cual se indique con son de sangre caliente y que sus crías se alimentan de leche que produce la madre y, después, podemos tener una rama donde se indique, por ejemplo, los mamíferos bípedos. Esta clase heredaría las características de los mamíferos y añadiría el hecho de que anden sobre las dos extremidades inferiores. Podríamos tener una hoja que sería el ser humano que heredaría las características de los mamíferos bípedos y añadiría las que hacen que el ser humano sea diferente de otros animales mamíferos y bípedos.

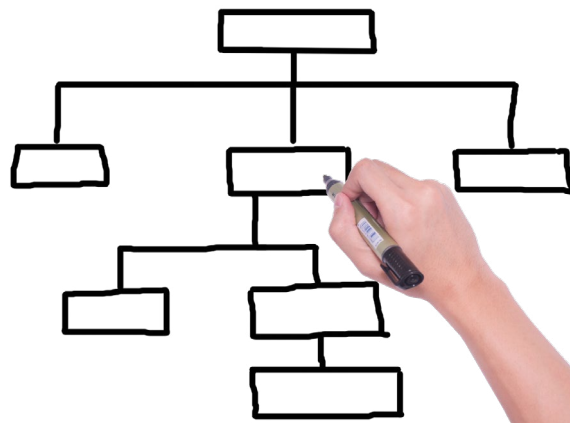


Figura 21 Jerarquía entre clases. Fuente: Pixabay, CC0 Creative Commons.

3.2.2. Conceptos OO

En la OO es muy importante diferenciar lo que es la Clase y los que es el Objeto.

Clase

Una clase es la descripción de todos los objetos de un mismo tipo. No podemos interactuar con una clase directamente sin antes instanciarla como objeto. Es una entidad abstracta.

Objeto

Instanciación de una clase. Es cuando asignamos a la clase valor a sus atributos con los de un objeto real de ese tipo, el cual queremos representar en nuestro programa para interactuar con él.

Veamos un ejemplo para fijar mejor los conceptos. Supongamos que queremos representar vehículos y para cada vehículo solo nos interesa el modelo, color y la cantidad de kilómetros que lleva recorridos.

- Ejemplo 1 Clase: Vehículo simple

class Vehiculo:

```
def __init__(self, pModelo, pColor, pKms):
```

```
    self.__modelo = pModelo
```

```
    self.__color = pColor
```

```
    self.__kms = pKms
```

```
def datosVehiculo(self):
```

```
    print('Modelo = ', self.__modelo, 'Color = ', self.__color, 'Kms = ', self.__kms)
```

```
vehiculoR = Vehiculo('F-150', 'Rojo', 10000)
```

Definimos la clase *vehiculo* con tres atributos privados: *modelo*, *color* y *kms* (fijarse que el nombre empieza por dos guiones bajos).

Tenemos una operación pública *datosVehiculo()* para mostrarnos los datos de una instancia (vehículo concreto).

En el ejemplo si hacemos:

```
vehiculoR.datosVehiculo() → Nos mostrará los datos: F-150, Rojo, 10000.
```

Si intentamos acceder directamente a la variable *modelo*:

```
print(vehiculoR.__modelo) → Obtendremos un error:  
AttributeError: 'Vehiculo' object has no attribute '__modelo'
```

No es que no lo tenga, sino que, al ser privado, "externamente" no existe y no podemos acceder directamente a él.

Ahora añadiremos un atributo adicional que será el precio de compra y una función que nos calculará el valor actual por si lo queremos vender.

- Ejemplo 2 Clase: Vehículo con valor

```
class VehiculoConValor:
```

```
    def __init__(self, pModelo, pColor, pKms, pPrecio):
```

```
        self.__modelo = pModelo
```

```
        self.__color = pColor
```

```
        self.__kms = pKms
```

```
        self.__precioCompra = pPrecio
```

```
    def datosVehiculoConValor(self):
```

```
        print('Modelo =', self.__modelo, 'Color =', self.__color, 'Kms =', self.__kms, 'Valor =', self.__valorActual())
```

```
    def __valorActual(self):
```

```
        return self.__precioCompra * ((100000-self.__kms)/100000)
```

```
vehiculoR = Vehiculo('F-150', 'Rojo', 10000, 20000)
```

```
vehiculoR.datosVehiculo()
```

En este segundo ejemplo podemos tener una función oculta para saber el valor actual del vehículo en función de los kms. Cada vez que llamemos a la función `datosVehiculo()`, se calculará usando internamente `__valorActual()` el valor actual del vehículo. Es útil si queremos que los compradores no sepan cómo se calcula el precio, por ejemplo.

También podríamos haber utilizado herencia para heredar las propiedades de `Vehiculo` en la clase `VehiculoConValor` y tendríamos dos funciones disponibles `datosVehiculoConValor()` y `datosVehiculo()`. Para hacerlo, debemos indicar la clase padre o raíz en la declaración.

```
class ClaseHijo(ClasePadre)
```


Tema 4.

Diseño modular y creación de bibliotecas

Hasta ahora, hemos estado trabajando en los ejercicios y ejemplos con **programación estructurada**, que es un paradigma de programación para mejorar la claridad, calidad y tiempo de desarrollo del software.

Dividiremos los programas en el **programa principal** (*main*) y **funciones**. Con la división de los problemas en subproblemas más manejables, al mismo tiempo que permitimos la reutilización, por ejemplo, si tenemos varios puntos donde debemos verificar que una fecha sea correcta, hacemos una función para eso (por ejemplo, una función que nos regrese Verdadero o Falso si la fecha es correcta con la firma *verificarFecha(fechaAVerificar)*), y la utilizamos en todos ellos.

En la figura 22 se puede observar cómo, en el ejemplo usado en el capítulo anterior para ilustrar una clase, tenemos dos partes claramente diferenciadas: el programa principal que, en este caso, es la creación de una instancia de la clase vehículo y la llamada a la función para saber los datos del vehículo y las funciones que son las operaciones que están implementadas en la clase.

Programación estructurada

Funciones, en este caso usando clases	<pre>class Vehiculo: def __init__(self, pModelo, pColor, pKms, pPrecio): self.__modelo = pModelo self.__color = pColor self.__Kms = pKms self.__precioCompra = pPrecio def datosVehiculo(self): print('Modelo =', self.__modelo, 'Color =', self.__color, 'Kms =', self.__Kms, 'Valor =', self.__valorActual()) def __valorActual(self): return self.__precioCompra * ((100000 - self.__Kms) / 100000)</pre>
Main	<pre>vehiculoR = Vehiculo('F-150', 'Rojo', 10000, 20000) vehiculoR.datosVehiculo()</pre>

Figura 22. Ejemplo de programa con la parte principal y funciones claramente diferenciadas. Fuente: Elaboración propia.

4.2. Diseño modular

Un paso más para hacer más manejables los softwares complejos es el paradigma de la **programación modular**. Consiste en **dividir** un programa en **módulos** o subprogramas. Para hacer esta división, deberemos abstraernos a diferentes niveles para permitirnos realizar mejor esta tarea. No pensaremos en los detalles específicos de implementación, sino que pensaremos primero en un alto nivel de abstracción como descomponer el problema y, solo cuando estemos en los niveles más bajos de abstracción, nos preocuparemos de cómo será la implementación.

Los problemas complejos se dividen en problemas más asequibles, el programa en subprogramas, aplicando el principio *divide et impera* acuñado por Julio César. Esta división se hará de forma recursiva (se dividirá las veces que sea necesario el problema), hasta conseguir un problema (programa) que puede ser abordado de forma “fácil” e implementar un código que lo solucione.

Los **módulos** son cada subprograma (o subsubprograma, subsubsubprograma, ...) que resuelve uno de los subproblemas en el que hemos dividido nuestro problema inicial.

Gráficamente, lo podemos ver como un muro (programa) formado por varios bloques (módulos). Cada módulo, de forma individual, soluciona un problema (tapa un hueco en el muro) y todos juntos con el programa principal (cemento) solucionan el problema (ver figura 23).

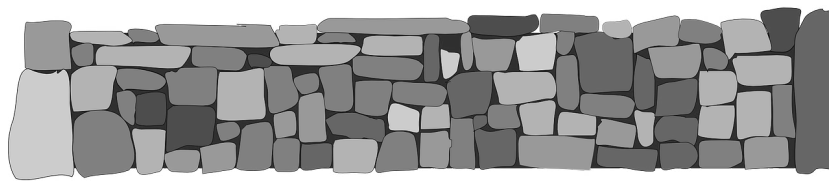


Figura 23. Muro formado por bloques. Fuente: Pixabay, CC0 Creative Commons.

Adicionalmente, al tener dividido el problema en subproblemas, puede ser que esas piezas o subproblemas se den en otras situaciones. Si, más adelante, nos piden resolver un problema que alguna de sus partes pueda ser descompuesta en una de las que ya hemos resuelto, podremos reutilizar el código/módulo. Así, reduciremos de forma notable el tiempo de desarrollo del programa para el "nuevo" problema. **Nos permite la reutilización**, en otras palabras, no hace falta inventar la rueda cada vez.

4.3. Diseño modular en Python

Para crear un **módulo** en Python, crearemos un **archivo donde** agruparemos todas las **definiciones y declaraciones que tengan relación**. Podemos también agrupar cosas sin relación, pero entonces no estaríamos aplicando diseño modular y dificultaría que este módulo fuese reutilizado.

Veamos un ejemplo con un módulo simple para implementar la sucesión de Fibonacci. Creamos archivo, el nombre del cual debe ser el nombre del módulo con el sufijo *.py*

fibonacci.py

```
# modulo Fibonacci

def sucFibonacci(k):

    if k == 1 or k == 2:

        result = 1

    else:

        result = sucFibonacci(k-1) + sucFibonacci(k-2)

    return result
```

Para poder **utilizar** el **módulo** creado deberemos **importar** el **módulo** a nuestro programa. Para hacerlo, indicaremos el nombre del módulo al principio del código utilizando la palabra clave *import* (en el ejemplo *import fibonacci*).

```
prog_principal.py

import fibonacci

seguir = True

while seguir:

    entrada = input('Fibonacci de ? (s para salir): ')

    if entrada != 's':

        print(fibonacci.sucFibonacci(int(entrada)))

    else:

        print('Adios')

        seguir = False
```

Si utilizamos `import "nombre del módulo"`, tendremos disponible en nuestro programa todo lo implementado en el módulo. A veces, si el módulo tiene muchos elementos y solo utilizaremos un pequeño grupo de ellos, es mejor importar exclusivamente lo necesario usando:

```
from modulo import elementos
```

De esta forma, solo importaremos lo que vayamos a utilizar. Por ejemplo, en el caso del módulo Fibonacci, si este tuviese más funciones y solo nos interesara la de calcular la serie, podríamos hacer:

```
from fibonacci import sucFibonacci
```

4.3.1. Packets

La **agrupación de módulos** en un mismo directorio, en el cual también creamos un fichero especial llamado `__init__.py`, es lo que convierte en esa carpeta en un **packet**.

El archivo `__init__.py` puede ser vacío.

Es una forma de organizar los módulos que soluciona una problemática específica. Por ejemplo, podríamos tener un *packet* matemático y, como parte de él, el módulo Fibonacci (ver figura 24).

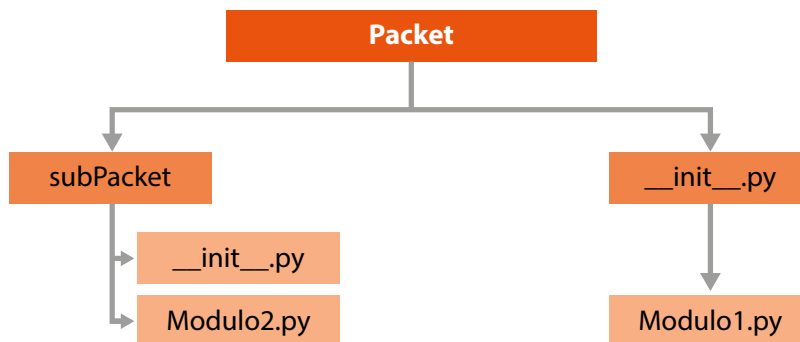


Figura 24. Ejemplo de Módulos Agrupados en un Packet. Fuente: Elaboración propia.

Hay infinidad de paquetes para Python, cuando tengamos nuestro problema descompuesto, es muy recomendable buscar primero si ya existe un módulo que lo resuelve total o parcialmente para reaprovechar los bloques que nos puedan ayudar a resolverlo sin necesidad de empezar de 0.

Por ejemplo, podemos buscar en el repositorio PyPI - *The Python Package Index*:

<https://pypi.python.org/pypi>

En el momento de la consulta cuando se elaboró este documento, teníamos más de 120.000 *packages* disponibles para su reutilización en nuestros programas (ver figura 25).



Figura 25. Captura de la web <https://pypi.python.org/pypi>. Fuente: <https://pypi.python.org/pypi>

Para poder navegar entre tantos paquetes, tenemos dos funcionalidades que nos lo facilitan. Podemos realizar una búsqueda directa *search* o podemos ir navegando por temas *topic* hasta localizar las funcionalidades que necesitamos (ver figura 26).

<https://pypi.python.org/pypi?%3Aaction=browse> (Python Software Foundation, Package Index, 2018)

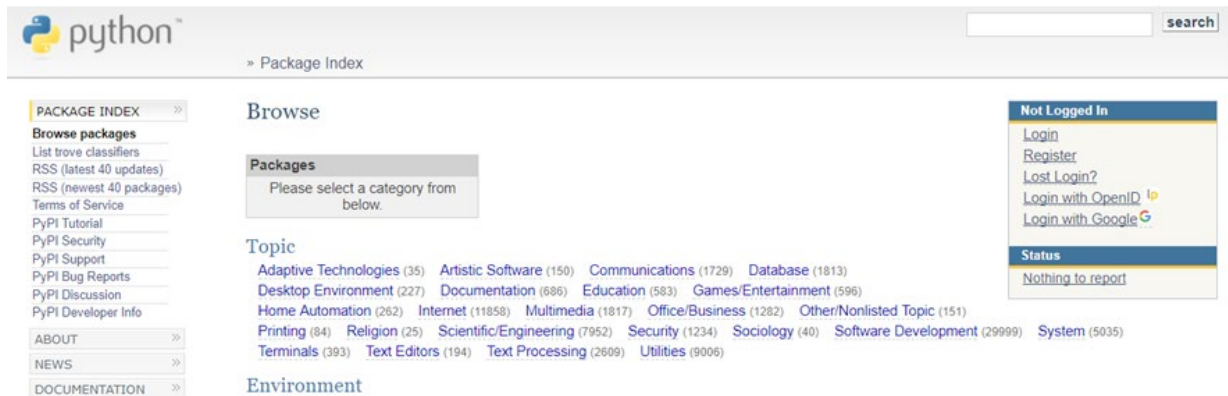


Figura 26. Búsqueda directa o navegar por temas (Python Software Foundation, Package Index, 2018).

Fuente: <https://pypi.python.org/pypi>

Por ejemplo, si estamos trabajando con historias clínicas electrónicas, disponemos del cliente FHIR (Fast Healthcare Interoperability Resources) ya implementado y listo para usarse. Dicho cliente nos permite acceder a varias funcionalidades de la historia clínica electrónica utilizando el estándar FHIR sin necesidad de que las implementemos de nuevo. Podemos acceder al packet de <https://pypi.python.org/pypi/fhirclient> y descargarlo en nuestro ordenador para su uso (ver figuras 27 y 28).








Name	Size	Packed Size	Modified	Mode	User
 models	2 847 060	2 919 936	2017-04-13 13:59	0rwxr-xr-x	nshver01
 auth.py	15 492	15 872	2017-04-13 13:50	0rw-r--r--	nshver01
 client.py	8 737	9 216	2017-04-13 13:50	0rw-r--r--	nshver01
 fhirreference_tests.py	5 662	6 144	2017-04-13 13:50	0rw-r--r--	nshver01
 server.py	11 127	11 264	2017-04-13 13:50	0rw-r--r--	nshver01
 server_tests.py	2 805	3 072	2017-04-13 13:50	0rw-r--r--	nshver01
 __init__.py	139	512	2016-04-15 08:54	0rw-r--r--	nshver01

Figura 27. Ficheros de los módulos principales del packet FHIR Client (Python Software Foundation, Package Index, 2018).

Fuente: A flexible client for FHIR servers supporting the SMART on FHIR protocol, <https://pypi.python.org/pypi/fhirclient>

Add Extract Test Copy Move Delete Info						
C:\Users\Roger\Downloads\fhircient-3.0.0.tar.gz\dist\fhircient-3.0.0\tar\fhircient-3.0.0\fhircient\models\						
Name	Size	Packed Size	Modified	Mode	User	Group
account.py	7 244	7 680	2017-04-13 13:50	0rw-r--r--	nshver01	staff
account_tests.py	5 117	5 120	2017-04-13 13:50	0rw-r--r--	nshver01	staff
activitydefinition.py	14 610	14 848	2017-04-13 13:50	0rw-r--r--	nshver01	staff
activitydefinition_tests.py	29 446	29 696	2017-04-13 13:50	0rw-r--r--	nshver01	staff
address.py	3 167	3 584	2017-04-13 13:50	0rw-r--r--	nshver01	staff
adverseevent.py	9 130	9 216	2017-04-13 13:50	0rw-r--r--	nshver01	staff
adverseevent_tests.py	2 072	2 560	2017-04-13 13:50	0rw-r--r--	nshver01	staff
age.py	795	1 024	2017-04-13 13:50	0rw-r--r--	nshver01	staff
allergyintolerance.py	9 085	9 216	2017-04-13 13:50	0rw-r--r--	nshver01	staff
allergyintolerance_tests....	4 397	4 608	2017-04-13 13:50	0rw-r--r--	nshver01	staff
annotation.py	2 135	2 560	2017-04-13 13:50	0rw-r--r--	nshver01	staff
appointment.py	8 713	9 216	2017-04-13 13:50	0rw-r--r--	nshver01	staff
appointmentresponse.py	3 536	3 584	2017-04-13 13:50	0rw-r--r--	nshver01	staff
appointmentresponse_t...	3 023	3 072	2017-04-13 13:50	0rw-r--r--	nshver01	staff
appointment_tests.py	9 470	9 728	2017-04-13 13:50	0rw-r--r--	nshver01	staff
attachment.py	2 505	2 560	2017-04-13 13:50	0rw-r--r--	nshver01	staff
auditevent.py	13 760	13 824	2017-04-13 13:50	0rw-r--r--	nshver01	staff
auditevent_tests.py	28 259	28 672	2017-04-13 13:50	0rw-r--r--	nshver01	staff
backboneelement.py	1 456	1 536	2017-04-13 13:50	0rw-r--r--	nshver01	staff
basic.py	2 786	3 072	2017-04-13 13:50	0rw-r--r--	nshver01	staff
basic_tests.py	5 920	6 144	2017-04-13 13:50	0rw-r--r--	nshver01	staff
binary.py	1 768	2 048	2017-04-13 13:50	0rw-r--r--	nshver01	staff
binary_tests.py	1 093	1 536	2017-04-13 13:50	0rw-r--r--	nshver01	staff
bodysite.py	3 145	3 584	2017-04-13 13:50	0rw-r--r--	nshver01	staff
bodysite_tests.py	4 282	4 608	2017-04-13 13:50	0rw-r--r--	nshver01	staff
bundle.py	11 073	11 264	2017-04-13 13:50	0rw-r--r--	nshver01	staff
bundle_tests.py	30 543	30 720	2017-04-13 13:50	0rw-r--r--	nshver01	staff
capabilitystatement.py	35 158	35 328	2017-04-13 13:50	0rw-r--r--	nshver01	staff
capabilitystatement_tes...	12 586	12 800	2017-04-13 13:50	0rw-r--r--	nshver01	staff

Figura 28. Todos los módulos para trabajar con FHIR (Python Software Foundation, Package Index, 2018). Fuente: A flexible client for FHIR servers supporting the SMART on FHIR protocol, <https://pypi.python.org/pypi/fhircient>

Tema 5.

Herramientas de depuración, pruebas y validación

En este tema, veremos algunos conceptos para mejorar la calidad de nuestros programas. Es importante señalar que la teoría solo no basta, es muy importante la práctica, para que los conceptos sean realmente útiles es fundamental usarlos para la depuración, prueba y validación del proyecto asignado para el semestre.

5.1. Herramientas de depuración

Depurar un programa consiste en **encontrar y corregir los problemas** que pueda tener el código del mismo. En inglés, *Debug*.

Podemos hacerlo por **métodos “arcaicos”** usando una herramienta básica como es la función *print()* para ir viendo los valores de las variables relevantes del programa, o ir imprimiendo texto por pantalla cada cierto número de instrucciones (sobre todo cuando las instrucciones llevan a una bifurcación como un *if else*) para saber por qué ramas del código transcurre la ejecución y qué valores van tomando las diferentes variables.

También contamos con **herramientas más poderosas** como es el **Depurador**, que nos ayuda a detectar y corregir los fallos más rápidamente al permitir la ejecución interactiva del programa. Como

inconveniente del Depurador, indicar que ralentiza la ejecución del código haciendo que un simple programa pueda durar varios minutos, pero hay técnicas para minimizar este punto. Utilizarlo nos **proporciona una información muy útil** que compensa con creces este inconveniente, hay errores que son muy difíciles de detectar y corregir con el uso exclusivamente de la función `print()`.

5.1.1. `print()`

Siendo una de las formas más simples, a veces es suficiente para poder detectar el error y corregirlo. Veamos un ejemplo de uso con la sucesión de Fibonacci que ya hemos visto anteriormente.

```
def fibonacci(k):  
    if k == 1 or k == 2:  
        result = 1  
    else:  
        result = fibonacci(k-1) + fibonacci(k-2)  
    return result  
  
f = int(input('Fibonacci de ? '))  
  
print('Fibonacci(', f, ') = ', fibonacci(f))
```

Al ser implementado en esta ocasión con una función recursiva (que se llama a sí misma), podemos querer ver cómo funciona paso a paso para entenderla mejor añadiendo algunas funciones `print()` que nos muestren un texto distintivo y el valor de alguna de las variables en puntos estratégicos.

```
def sucFibonacci(k):  
    # para saber el valor que recibe la función  
  
    print('k = ', k)  
  
    if k == 1 or k == 2:  
        result = 1  
    else:  
        # Para saber los valores que pasamos a las dos llamadas  
        # recursivas  
  
        print('llamada recursiva, k-1 = ', k-1, 'k-2 = ', k-2)  
  
        result = sucFibonacci(k-1) + sucFibonacci(k-2)
```

```
# Para saber el valor retornado por la función

print('return = ', result)

return result

f = int(input('Fibonacci de ? '))

print('Fibonacci(',f,') = ',sucFibonacci(f))
```

En la figura 29 podemos ver el resultado de ejecutar el programa que calcula Fibonacci con los diferentes `print()` intercalados.

```
Fibonacci de ? 5
k = 5
llamada recursiva, k-1 = 4 k-2 = 3
k = 4
llamada recursiva, k-1 = 3 k-2 = 2
k = 3
llamada recursiva, k-1 = 2 k-2 = 1
k = 2
return = 1
k = 1
return = 1
return = 2
k = 2
return = 1
return = 3
k = 3
llamada recursiva, k-1 = 2 k-2 = 1
k = 2
return = 1
k = 1
return = 1
return = 2
return = 5
Fibonacci( 5 ) = 5
```

Figura 29. Resultado de la ejecución con los `print()`. Fuente: Elaboración propia.

5.1.2. Spyder Debugger

Una herramienta mucho más potente es utilizar el *debugger*, el depurador. Con ella, podremos ejecutar el programa paso por paso de forma interactiva e ir viendo, **instrucción por instrucción**, por dónde pasa la ejecución y consultar en vivo los valores de las variables. En la figura 30 podemos ver la interfaz gráfica del depurador integrado con el programa Spyder, el entorno de desarrollo para programación científica en Python de software libre que utilizaremos para el desarrollo de nuestros programas durante la asignatura.

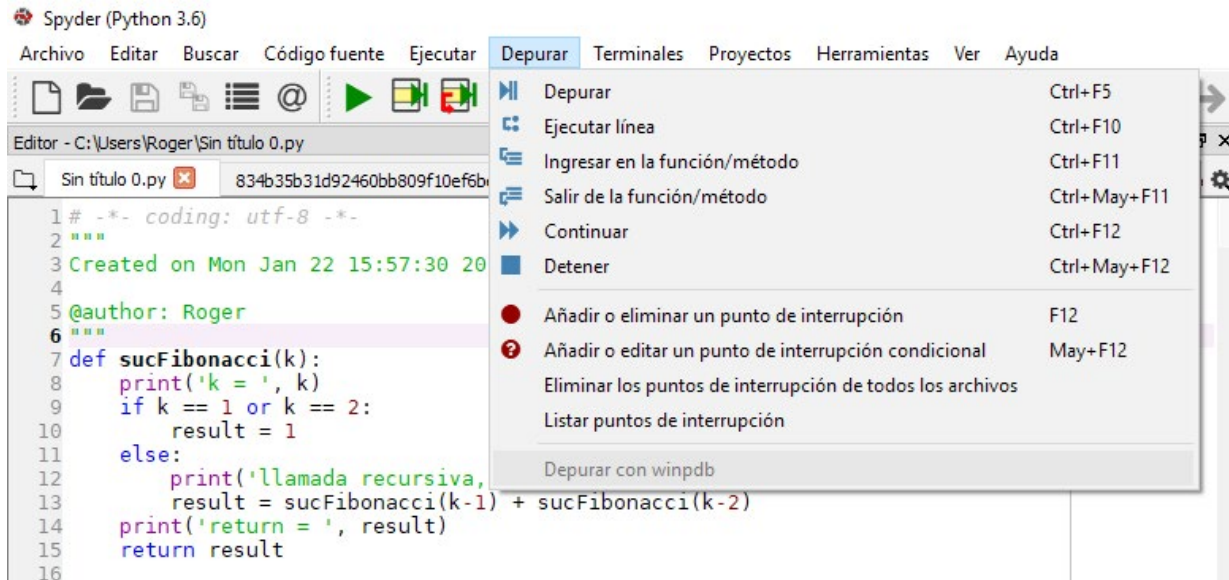


Figura 30. Depurador del Spyder (Python Software Foundation, Spyder, 2018).

Fuente: Elaboración propia, captura de pantalla del programa Spyder.

Accedemos a la interfaz gráfica utilizando la opción del menú superior "Depurar", como vemos en la figura 30. En primer lugar, siempre debemos empezar por la opción "Depurar" para ejecutar el programa en modo *debug* y, a partir de allí, podemos ir avanzando paso a paso con la opción "Ejecutar línea".

Si ejecutamos una línea donde hay una llamada a una función, por defecto se ejecutará esta completamente. Si queremos entrar en la función y ver paso a paso cómo se ejecuta su código, debemos usar la opción "Ingresar en la función/método". Si una vez dentro de una función nos cansamos o no es de interés su contenido para detectar el problema, podemos ejecutar todo el código de la función hasta regresar a la llamada con la opción "Salir de la función/método".

Una manera más óptima es ejecutar de forma continua el código hasta encontrar un punto de interrupción (*breakpoint*). Para eso debemos, en primer lugar, colocar en el código los puntos de interrupción en los lugares de interés usando la opción "Añadir o eliminar un punto de interrupción", una sola vez añade un nuevo punto de interrupción. Si realizamos la acción de añadir en un punto donde ya existía uno, lo elimina.

Una forma más avanzada todavía es que el punto de interrupción sea condicional con la opción "Añadir o eliminar un punto de interrupción condicional", igual que en el caso anterior, si ya existía, lo elimina. En el caso de tener puntos de interrupción marcados en nuestro código, en vez de usar la ejecución paso a paso, usaremos la opción "Continuar" donde la ejecución se hará de forma interrumpida hasta conseguir un punto de interrupción donde podremos cambiar paso a paso o volver a darle a continuar para que siga ejecutando hasta el siguiente punto de interrupción (ver figura 31).

Para finalizar en cualquier momento la ejecución del programa actual en modo *debug*, tenemos la opción de "Detener".



Figura 31. Ejemplo Depurador gráfico del Spyder (Python Software Foundation, Spyder, 2018).

Fuente: Elaboración propia, captura de pantalla del programa Spyder.

Otra opción es usar el *debug* por línea de comandos, que son análogos a los que podemos usar gráficamente:

n (*next*) → Siguierte instrucción. No entraremos en las funciones, se ejecutarán como una sola línea de código, debemos usar *step into* para entrar en ellas.

p (*print*) → Imprime el valor de la variable. Uso: `p nombre_variable`. La variable también puede ser solo una posición específica de una estructura más compleja, por ejemplo, `p nombre_vector[posición]`.

q (*quit*) → Salir del programa

c (*continue*) → Deja de depurar y sigue la ejecución normal del programa hasta el siguiente punto de interrupción o hasta finalizarlo completamente.

l (*list*) → Muestra el pedazo de código por el que va la ejecución, nos ayuda a situarnos en programas grandes.

s (*step into*) → Entra en la función para poder verificar si en ella hay un error.

r (*resume*) → Ejecutar hasta salir de la función actual.

En la figura 32 podemos observar un ejemplo de depuración por línea de comandos.

```

In [8]: debugfile('C:/Users/roger.clotet/AppData/Local/Temp/vWorkspace/fibonacci.py',
wdir='C:/Users/roger.clotet/AppData/Local/Temp/vWorkspace')
> c:\users\roger.clotet\appdata\local\temp\vworkspace\fibonacci.py(4)<module>()
      2
      3
----> 4 def fibonacci(k):
      5     if k == 1 or k == 2:
      6

ipdb> n
> c:\users\roger.clotet\appdata\local\temp\vworkspace\fibonacci.py(14)<module>()
     12     return result
     13
--> 14 f = int(input('Fibonacci de ? '))
     15
     16 print('Fibonacci(', f,') = ', fibonacci(f))

ipdb> n
Fibonacci de ? 4
> c:\users\roger.clotet\appdata\local\temp\vworkspace\fibonacci.py(16)<module>()
     12     return result
     13
     14 f = int(input('Fibonacci de ? '))
     15
--> 16 print('Fibonacci(', f,') = ', fibonacci(f))

ipdb> p f
4

ipdb> q
Fibonacci( 4 ) =  3

In [9]: |

```

Figura 32. Ejemplo Depurador por línea de comandos del Spyder (Python Software Foundation, Spyder, 2018).
Fuente: Elaboración propia, captura de pantalla del programa Spyder.

5.2. Pruebas y Validación

Hace más de 30 años que empezó una preocupación a nivel global sobre la cantidad de recursos que se desperdiciaban en software que no cumplía con lo esperado. Se calculaban pérdidas de millones de dólares anualmente, aun cuando se empezaron a tomar medidas para mitigar el problema y se siguen tomando, en la actualidad aún se calcula que los costes asociados se cuentan por millones anualmente.

Una de las principales medidas que podemos utilizar como Ingenieros Informáticos para minimizar estas pérdidas es incluir dentro del proceso de creación del software la realización de **pruebas** para garantizar que el mismo cumpla con lo esperado (**validación**) y, al mismo tiempo, también minimizar los errores que este pueda tener. Los errores, por pequeños que sean, pueden provocar grandes pérdidas. Según los estudios, con solo 3 o 4 errores por cada 1000 líneas de código provocan que el

código sea defectuoso y que no funcione como se esperaba, provocando cuantiosas pérdidas a sus usuarios (Pressman 2010).

Para la validación, es importante realizar pruebas siguiendo dos aproximaciones, pueden hacerse **por bloques** o a todo el **software como un conjunto**. Es recomendable hacerlo tanto de una manera como de la otra, ya sea primero por bloques y después en conjunto o viceversa. Realizarlos **por bloques nos permite aislar** mejor los posibles errores y es más fácil detectar la parte del código implicada y proceder a su solución. Al mismo tiempo, también es bueno hacer la prueba como un solo **conjunto** para comprobar la **correcta integración** de los diferentes **componentes**, ya que cada uno funcione correctamente no significa que todos juntos lo hagan.

Pensemos en el ejemplo de un vehículo, si ensamblamos el cambio de marchas de forma incorrecta con el motor (con los engranajes al revés) podemos provocar que este solo tenga una marcha hacia delante y diversas hacia atrás por mucho que, por separado, el motor y la caja de cambios funcionen de forma correcta (el motor genera fuerza de tracción y el cambio de marchas cambia entre las diferentes relaciones).

Las pruebas por bloques requieren ir probando de forma individual los módulos/componentes. Puede ser necesario realizar un programa principal especial que se encargue exclusivamente de llamar a todas las funciones del módulo/componente. Para realizar la prueba en conjunto se utiliza el programa principal que hemos elaborado casi sin modificaciones, a veces serán necesarios pequeños cambios para agilizar el proceso.

Adicionalmente, si hacemos las pruebas por bloque o como un conjunto, tenemos dos métodos posibles para realizar las pruebas en ambos casos. Podemos realizar la ejecución a ciegas, lo que se conoce como caja negra, o con pleno conocimiento del código, lo que se conoce como caja blanca. Las dos aproximaciones son complementarias entre ellas como en el caso de por bloques o como un solo conjunto (ver figura 33).

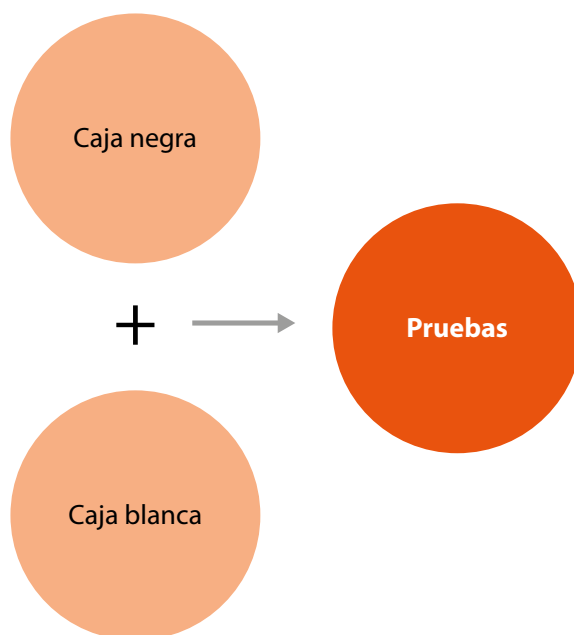


Figura 33. Dos aproximaciones complementarias, caja negra y caja blanca. Fuente: Elaboración propia.

En casi todos los contratos para la elaboración de un software, el cliente acostumbra a cubrirse las espaldas en cuanto a la calidad y correcto funcionamiento del producto final exigiendo un mínimo de pruebas para aceptar el software como apto para aceptarlo y ponerlo en producción.

A continuación, veremos con un poco más de detalle los dos métodos: Caja Blanca y Caja Negra.

5.2.1. Caja Negra

No sabemos cómo es la **implementación interna**, solo verificamos que el interfaz del programa cumpla con los requisitos funcionales del software. Puede haber líneas de código inservibles o erróneas que no comprobemos al solo fijarnos en los requisitos.

En una función de un **conjunto controlado de valores de entrada**, se **verifica** que **las salidas producidas** por el software **sean las esperadas** (ver figura 34).



Figura 34. Caja Negra. Fuente: Elaboración propia.

Las pruebas de caja negra nos ayudan a detectar los tipos de errores siguientes:

- **Funciones con comportamiento incorrecto o no implementadas**

Dada una entrada, se obtiene una salida incorrecta o simplemente para una determinada entrada no prevista por los programadores, el sistema no tiene implementado cómo responder.

- **Errores de interfaz**

Ya sea en la interfaz con el usuario o con otros programas, está mal implementada. Entre otros, nos podemos encontrar errores de no permitir acceder a todas las funciones, no pedir todos los datos necesarios para llamarlas correctamente o tener mal configurado el mapeo de los valores provocando el paso incorrecto de estos.

- **Errores en las estructuras o en cómo se guardan los datos**

Podemos encontrar que las estructuras diseñadas para guardar los datos estén incorrectamente diseñadas o que la base de datos necesita algunos parámetros de configuración adicionales en las tablas, como puede ser la definición de las claves primarias que determina qué atributos son los identificadores para cada elemento de una tabla. Por ejemplo, el sistema no debería dejarnos dar de alta a dos personas con el mismo documento de identidad.

- **Errores de comportamiento o rendimiento**

El comportamiento debe ser siempre como se ha indicado en las especificaciones para considerarlo correcto, adicionalmente este debe ser consistente (a igual entrada, igual salida siempre).

El rendimiento es un parámetro a tener en cuenta en muchos contextos, hay funcionalidades que son extremadamente sensibles al tiempo de respuesta. Un ejemplo extremo puede ser un sistema de defensa antimisiles, si el sistema tarda demasiado tiempo en determinar si un misil detectado es una amenaza o no y en calcular qué baterías de misiles interceptores deben neutralizarlo, podemos encontrarnos que el misil impacte antes que pueda ser neutralizado o que las probabilidades de neutralizarlo sean muy bajas.

- **Errores de inicialización y terminación**

En determinadas circunstancias, el programa no consigue levantarse (inicializarse) correctamente. O cuando salimos de él, hay veces que no se guarda correctamente el estado provocando la pérdida de información o que después no se inicialice correctamente.

A diferencia de las pruebas de caja blanca que se pueden realizar en las primeras fases de desarrollo, por ejemplo, cuando un componente ya está listo sin esperar a tener todo el programa, las pruebas de caja negra necesitan tener el desarrollo avanzado y **acostumbran a realizarse en las fases finales de desarrollo**. Las pruebas deben ser diseñadas para ayudarnos a responder a las siguientes preguntas (Pressman, 2010):

- ¿Cómo se prueba la validez funcional?

Hace lo que debe hacer.

- ¿Cómo se prueban el comportamiento y el rendimiento del sistema?

Se comporta tal como debería.

Hace las funciones consumiendo los recursos previstos y en el tiempo estimado/requerido.

- ¿Qué clases de entrada harán buenos casos de prueba?

Pensar con detenimiento las entradas, cuáles son los valores normales y, más importante, cuáles son los valores extremos que deben verificarse.

- ¿El sistema es particularmente sensible a ciertos valores de entrada?

Tenemos, por ejemplo, casos específicos que hacen que el sistema se comporte mucho más lento de lo normal.

- ¿Cómo se aíslan las fronteras de una clase de datos?

Qué valores son incorrectos para determinados parámetros, por ejemplo, fecha de nacimiento de una persona mayor a la fecha actual o capacidad de un tanque de combustible negativo.

- ¿Qué tasas y volumen de datos puede tolerar el sistema?

Forzar el sistema hasta el límite para saber cómo reacciona en casos extremos e indicarlo como límite en la documentación de entrega o si debe soportarlo mejorar la implementación o asignación de recursos para que funcione sin inconvenientes en estos casos.

- ¿Qué efecto tendrán sobre la operación del sistema algunas combinaciones específicas de datos?

Los sistemas normalmente son multiusuario o concurrentes con la interfaz con otros programas, puede darse el caso que entre diferentes interacciones simultáneas con el sistema provoquen situaciones de bloqueo o que lleven a error.

5.2.2. Caja Blanca

Conociendo la implementación interna, tenemos acceso a todo el código y, en función de este, creamos los datos y casos de prueba. Estamos verificando centrados en la implementación, podemos obviar que el programa entregado no cumple alguno/s de lo/s requisito/s (ver figura 35).

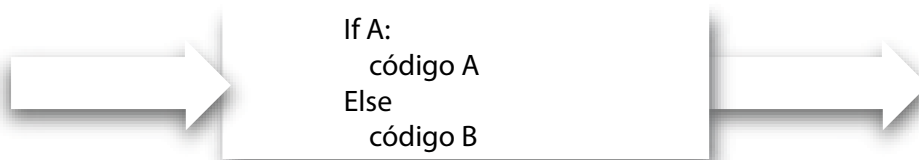


Figura 35. Caja Blanca. Fuente: Elaboración propia.

Las pruebas de caja blanca deben garantizar:

- Que todas las líneas de código, con sus diferentes líneas de ejecución dentro de un módulo, se revisaron al menos una vez.
- Cuando nos encontramos con expresiones de bifurcación que depende de un resultado lógico, probaremos los dos posibles resultados: Verdadero y Falso.
- En las estructuras iterativas (*while*, *for*) comprobaremos que funcionan correctamente tanto para los casos que entremos como en los que no y, en el caso de entrar en la estructura iterativa, comprobaremos que se ejecuta hasta que la condición frontera hace que salgamos de ella.
- Comprobaremos que las estructuras de datos son correctas, analizando su estructura interna.

5.2.3. Utilidad

Podemos considerar la utilidad de las pruebas en sus diferentes vertientes pues permiten:

Aceptación

Como hemos comentado anteriormente, casi todas las empresas las requieren antes de aceptar un software y pagar al proveedor, pues quieren garantizarse que el producto cumple con lo pedido exigiendo un mínimo de pruebas que este debe superar para su aceptación. En muchas ocasiones, exigen que estas sean realizadas por terceros especializados, pues muchas veces el cliente no cuenta con la experiencia o los recursos necesarios y tampoco es conveniente que el proveedor sea, al mismo tiempo, el que verifique, por evidente conflicto de interés.

Instalación

Validar que el software se instala correctamente en el entorno de producción, que puede variar ligeramente del de pruebas, diferente versión del sistema operativo o diferencias en el hardware. Aun cuando, en teoría, el software debería ser compatible sin muchas dificultades en el entorno de despliegues, si se ha utilizado un entorno similar para el desarrollo, esto muchas veces no es así. Pequeñas diferencias en las versiones de las librerías interna utilizadas, la presencia de un software que no estaba en el entorno de desarrollo, diferente idioma en la interfaz de usuario del sistema operativo, entre otras diferencias, pueden provocar que la instalación se convierta en un auténtico infierno. **Idealmente, se debe verificar** que esta se produce de forma correcta en un **entorno idéntico al de producción**, a veces no es posible por el elevado coste de tener el entorno duplicado para la realización exclusivamente de pruebas.

Funcional

Que hace lo que se espera de él, que cumple con los requisitos funcionales establecidos cuando se planificó. Aunque parezca extraño, uno de los principales motivos de que el software no sea usado y que su desarrollo sea un fracaso, es que en muchas ocasiones no cumple la funcionalidad para la que fue construido, ya sea porque simplemente no la cumple o porque no se adaptó a los cambios que pidieron los usuarios durante su desarrollo y despliegue.

Confiabilidad

Garantizar que el software hace de forma **confiable** su cometido, **sin** cometer **errores**. Los cálculos internos son ejecutados sin errores y de forma consistente (siempre con el mismo resultado, para una misma entrada de datos). Por ejemplo, pequeños errores de aproximación en cálculos pueden llevar a pérdidas millonarias para las empresas. Otro ejemplo es el área de seguridad, las funciones que calculan números aleatorios deben ser realmente aleatorios o estarían facilitando el trabajo a los hackers para romper la seguridad.

Rendimiento

Someter el software a una prueba de rendimiento (*stress*) que permite asegurar que este será capaz de soportar de forma eficaz la carga de trabajo en las **peores circunstancias de operación**. Para

realizar este tipo de pruebas de forma correcta, se necesita una cantidad de recursos considerables, ya que no se trata de, simplemente llevar al límite el sistema, sino también de hacerlo de una forma lo más parecida posible al uso real para que sea realmente útil. No sirve de nada, por ejemplo, saber que el sistema aguantará 100.000 peticiones simples por segundo si después el problema está en que es incapaz de soportar, por ejemplo, solo 10 peticiones complejas. Se utiliza software especializado para **simular** el **comportamiento** complejo de varios **usuarios reales** interactuando de forma concurrente con el sistema.

Usabilidad

Que el software sea cómodo para los usuarios. Por ejemplo, si hay una operación que se usa diariamente 1000 veces, que no sea necesario acceder a 4 menús para llegar a ella. En otras palabras, que les ayude a hacer su trabajo en vez de ponérselo más complicado.

Es un aspecto que, como programadores, a veces no tenemos en cuenta, muchas veces porque tampoco se nos proporciona la información necesaria de cómo será usado el sistema por los diferentes usuarios. Por este motivo, es interesante incorporar a los usuarios en el proceso de desarrollo y, sobre todo, en la parte de pruebas para corregir cuanto antes los inconvenientes de usabilidad que pueden llegar a provocar que un buen software no se use por ser demasiado complejo o lento de utilizar.

5.2.4. ¿Cómo?

Las pruebas y validación deben ser realizadas con una buena **planificación** para garantizar que son efectivas. Según las circunstancias de cada proyecto deberemos considerar varias variables como:

- Costo/Beneficio

Realizar las pruebas tiene un costo elevado (por ejemplo las de *stress* o las de instalación), en algunos desarrollos no será justificable hacer todas las posibles y en toda su profundidad. Deberemos balancear la relación coste/beneficio según el contexto de nuestro proyecto.

- Tiempo

Las pruebas son necesarias pero requieren de tiempo, casi siempre el tiempo requerido para realizar todas las pruebas será más del disponible y deberemos centrarnos en aquellas que sean más relevantes para el desarrollo actual.

- Alcance

Relacionado con los dos primeros puntos de Coste y Tiempo, muchas veces deberemos limitar el alcance de las mismas hasta donde estos dos recursos fundamentales nos permitan.

- Reusabilidad

Si estamos desarrollando diferentes softwares relativamente parecidos, es recomendable pensar las pruebas para que puedan ser reutilizadas para validar y probar los diferentes

software de la misma línea sin cambios o con los mínimos, o para que sean utilizadas para ir verificando las diferentes evoluciones del mismo producto.

- Automáticas/Manuales

Automatizar las pruebas tiene normalmente un coste más elevado que realizarlas manualmente. Pero debemos valorar los puntos anteriores, sobre todo la reusabilidad, que puede hacer que una inversión inicial más grande en automatizar algunas pruebas sea rápidamente amortizada. También hay algunas pruebas, como por ejemplo las de rendimiento, que son difíciles de realizar de forma manual y que no queda otra que automatizarlas.

Resumiendo, deberemos realizar una planificación de las pruebas a realizar. Idealmente deberíamos probar todo, pero casi siempre tenemos limitaciones de **tiempo** o **presupuesto** que hacen que el **alcance** de las pruebas se vea reducido y nos tengamos que circunscribir a garantizar un mínimo de verificación que nos proporcione la mejor relación **esfuerzo/beneficio** dentro de los límites legales del contrato de desarrollo firmado.

5.2.5. Evaluar resultados

Una vez realizadas las pruebas, deberemos evaluar los resultados obtenidos para dar el OK al software o, de ser necesario, realizar las correcciones necesarias en el mismo. De realizar correcciones, deberemos volver a realizar las pruebas en la nueva versión. Realizaremos el **proceso de forma iterativa hasta que el software supere las pruebas** y pueda ser entregado para su puesta en producción.

Tema 6.

Gestión de errores

Los errores en el software tienen un fuerte **impacto económico**, según la firma Tricentis dedicada a la verificación de software, de aproximadamente 1 billón de \$ anual (en 2016).

Aun cuando hagamos correctamente la validación y las pruebas siguiendo lo visto en el tema anterior, los programas, cuando estén en producción, afrontarán situaciones no previstas ni verificadas que pueden conllevar errores, más o menos graves, según el contexto donde se ejecute el mismo.

Por este motivo, es importante tanto detectar los errores y corregirlos en el código dentro en la fase de validación, como hacer que, ante situaciones inesperadas, el programa sepa cómo reaccionar (o como mínimo lo haga de la forma menos dañina posible), haciendo una correcta gestión de los errores en tiempo de ejecución para minimizar los inconvenientes. Cuando programamos, no debemos suponer que siempre las cosas saldrán bien, también pueden salir mal y, ante esta situación, habrá que tomar precauciones.

Algunos casos reales:

- ExoMars (ESA - 2016): el programa interpretó mal las lecturas de los sensores. Estos indicaron valores anormalmente altos debido a grandes perturbaciones en la entrada a la atmósfera de

Marte y el software lo interpretó como que ya estaba en el suelo soltando el paracaídas y desplegando la sonda. El resultado fue la destrucción de la misma al estrellarse a gran velocidad en el suelo marciano.

- Año 2000: usar solo los 2 últimos dígitos para indicar el año. Cuando fueron necesarios más dígitos para representar los años, regresaban al 1900 o daban errores.
- Therac-25: error de software en un equipo de radioterapia, permitía sobreexposición a la radiación con efectos mortales. No tenía en cuenta algunos factores y en determinadas circunstancias sobreexponía a los pacientes.
- *Knight Shows How to Lose \$440 Million in 30 Minutes*: software que automáticamente vendía y compraba acciones, pero lo hacía de forma poco beneficiosa para los clientes en determinadas circunstancias.

Más información: <https://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes>

6.1. Control de parámetros

Uno de los principales puntos donde se pueden producir errores es en el paso de **parámetros incorrectos a las funciones**. Ya sea en las llamadas internas dentro del código o cuando estos parámetros vienen de fuentes externas, por ejemplo, un archivo de configuración o son introducidos por los usuarios.

Ya sea con mala intención o por error, debemos contemplar la posibilidad de que los parámetros no sean los esperados y reaccionar de forma controlada ante esa eventualidad. Aun cuando en el desarrollo actual el programador realice las comprobaciones necesarias de los parámetros, puede ser que cuando la función sea reutilizada en otro contexto el programador no lo haga. Siempre es recomendable que la misma función verifique sus propios datos de entrada.

Por ejemplo, si tenemos un programa que administra una gasolinera y en él debemos cargar los datos de configuración de los tanques de combustible en una estructura de diccionario del tipo: `{'combustible': [stock actual, capacidad máxima]}`, podríamos hacer un recorrido del diccionario revisando que siempre el stock actual sea menor o igual a la capacidad máxima (ver figura 36).


```
7
8 # Configuración Inicial de la Gasolinera. Debe retornar True si todo va
  bien y False si hay algún tipo de error (para el ejemplo solo verifico
  que cantidad sea <= a capacidad)
9 def configInicial(diccionario_combustible_cantidad_capacidad):
10     """
11     Variable global que contendrá la configuración de la gasolinera.
12     Global para que la podamos inicializar en esta función y después
13     actualizarla durante la ejecución del programa des del main (programa
14     principal)
15     """
16     global gasolinera
17
18     # Verificar para todos los combustibles que cantidad es <= a la capacidad
19     ok = True # para controlar que todo este OK
20     """
21     En cada iteración del for tratamos un tipo de combustible (una
22     posición del diccionario)
23     El for tienen la ventaja que se encarga de empezar por la primera
24     posición (en el caso del diccionario la primera, no siempre sería la
25     que este pasada como primera, tienen un orden propio) y recorrerlas
26     todas de forma "automática".
27     Hay otras maneras de hacerlo, que en ocasiones pueden ser más
28     eficientes, por ejemplo con un While que controle que siga siendo ok
29     (al detectar un solo caso incorrecto ya podemos salir)
30     """
31     for combustible in diccionario_combustible_cantidad_capacidad:
32         #combustible es un tipo combustible
33         #diccionario_combustible_cantidad_capacidad[combustible] es el vector
34         [cantidad, capacidad máxima] para ese combustible
35
36         print(combustible, 'Stock = ', diccionario_combustible_
37               cantidad_capacidad[combustible][0], ' Max = ', diccionario_
38               combustible_cantidad_capacidad[combustible][0])
39         #Combustible[0] es [cantidad, capacidad máxima]
40         if diccionario_combustible_cantidad_capacidad[combustible][0]>
41             diccionario_combustible_cantidad_capacidad[combustible][1]:
42             ok = False
43
44     if ok:
45         #Todo OK, la gasolinera es inicializada
46         gasolinera = diccionario_combustible_cantidad_capacidad
47         return True
48     else:
49         #Algún caso es incorrecto, no asignamos y regresamos Falso
50         return False
51
52
```

Figura 36. Verificar Diccionario. Fuente: Elaboración propia.

Otra posibilidad es que el usuario introduzca un dato incorrecto, como por ejemplo una opción del menú inexistente. Debemos contemplar que cualquier dato introducido que no corresponda con lo esperado debe ser detectado y descartado (ver figura 37).

```
96 #Repetir para cada tipo
97 elif tipo == 4:
98     print('Ha seleccionado gasolina98')
99     cantidad = int(input('Introduce la cantidad de combustible:'))
100     if (cantidad <= gasolinera['gasolina98'][0]):
101         print('Completado el Repostaje')
102         gasolinera['gasolina98'][0]=gasolinera['gasolina98'][0]-cantidad
103         print('Stock de',gasolinera['gasolina98'][0], 'litros de gasolina98')
104     else:
105         print('Solo pudimos servirle', gasolinera['gasolina98'][0])
106         gasolinera['gasolina98'][0]=0
107         print('Gasolina98 agotada')
108 #Si no es alguno de los combustibles disponibles, indicar error. Volver
    a preguntar = otra vuelta del While
109 else:
110     print('Seleccione un tipo de los disponibles')
```

Figura 37. Validar opción elegida. Fuente: Elaboración propia.

6.2. Excepciones en el código

Es **imposible controlar todos los** posibles **errores** en el código y, aun cuando los controlásemos todos, hay veces que una **excepción** puede provocar una interrupción abrupta del programa durante su ejecución.

Una excepción es un error poco frecuente, que se produce por un error o interrupción no esperada durante la ejecución del código. El sistema crea un objeto especial que contiene información "detallada" sobre el error y es pasado/lanzado para que el mismo programador en su código o, en caso contrario el sistema operativo, lo trate para intentar recuperarse y seguir con la ejecución. No siempre es posible recuperarse y, a veces, inevitablemente conlleva que el programa actual aborte, en casos más graves hasta puede provocar que el sistema operativo necesite reiniciarse.

En esos casos, podemos aplicar técnicas de administración de las excepciones. Lo que permite "controlar" la reacción del programa delante de estos eventos e intentar seguir con la ejecución del mismo o abortarlo de forma controlada si el error es grave.

Aunque hay veces que no es posible recuperarse e inevitablemente el programa abortara (terminará su ejecución. Un ejemplo que todos hemos experimentado son las excepciones en el sistema operativo Windows que no pueden recuperarse y nos llevan a la pantalla azul (ver figura 38), en algunas versiones de Windows se hace un volcado de memoria para intentar ayudar a saber cuál fue el problema. Aun con la información del volcado, es difícil, por no decir imposible, identificar el problema para la mayoría de los usuarios, por eso las versiones más actuales lo han sustituido por un simple mensaje de error con una carita triste (ver figura 39).

A problem has been detected and windows has been shut down to prevent damage to your computer.

The end-user manually generated the crashdump.

If this is the first time you've seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup options, and then select Safe Mode.

Technical information:

*** STOP: 0x000000E2 (0x00000000,0x00000000,0x00000000,0x00000000)

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further assistance.

Figura 38. Excepción irrecuperable en Windows XP. Fuente: Windows XP.

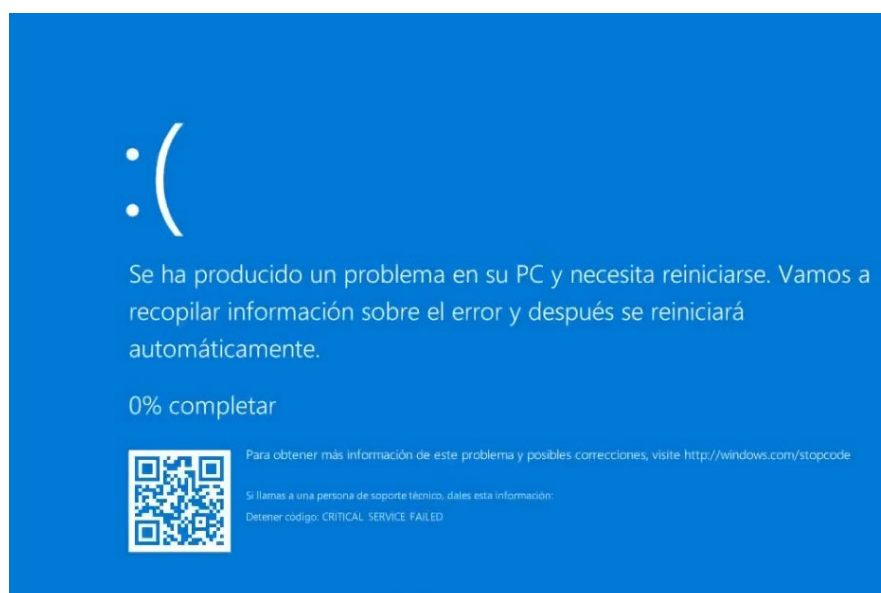


Figura 39. Excepción irrecuperable en Windows 10. Fuente: Windows 10.

6.3. Gestión de excepciones en Python

Python, como la mayoría de lenguajes de alto nivel, contempla la posibilidad de que el programador haga una gestión de las excepciones. Es responsabilidad del programador, pues el lenguaje por sí solo

no lo hace aunque ofrece algunas facilidades como son, por ejemplo, las excepciones predefinidas por defecto.

6.3.1. Sin gestión

Si no realizamos gestión de las excepciones en Python, nuestros programas **pueden terminar de forma abrupta y nada elegante**.

Por ejemplo, si nuestro programa intenta hacer una división por 0, operación irrealizable por una computadora estándar, obtendremos una excepción de *ZeroDivisionError* (ver figura 40). También, si intentamos acceder a una posición inexistente en una estructura de datos, por ejemplo, un vector de 2 posiciones y en cual queramos acceder a la posición *v[2]* (recordad que en el caso de Python, como en otros lenguajes, los vectores empiezan en la posición 0) (ver figura 41). O si intentamos sumar peras con manzanas que sería, en el caso de programación, intentar por ejemplo sumar enteros con cadenas de caracteres (ver figura 42).

```
In [1]:      10/0
Traceback (most recent call last):
  File "<ipython-input-1-242277fd9e32>", line 1, in <module>
    10/0
ZeroDivisionError: division by zero
```

Figura 40. Excepción *ZeroDivisionError*. Fuente: Elaboración propia.

Nombre	Tipo	Tamaño	Valor
v	list	2	[1, 2]

```
File "C:/Python/WinPython-64bit-3.4.4.5Qt5/settings/.spyder-py3/temp.py", line 10, in
<module>
    print(v[2])
IndexError: list index out of range
```

Figura 41. Excepción *IndexError*. Fuente: Elaboración propia.

```
In[6]:      7 + 'Siete'
Traceback (most recent call last):
  File "<ipython-input-6-4a67acce82e0>", line 1, in <module>
    7 + 'Siete'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figura 42. Excepción *TypeError*. Fuente: Elaboración propia.

Los programas funcionarán aun si no hacemos gestión de excepciones, pero es altamente recomendable hacerla siempre.

6.3.2. Excepciones Predefinidas

Por defecto, hay un **conjunto** de **excepciones predefinidas**, estas están clasificadas por tipos, el programador puede definir adicionales o redefinir el comportamiento de las que vienen por defecto. Por ejemplo, en el caso de división por 0 (*ZeroDivisionError*) que hemos visto en el apartado anterior.

Aparte del tipo de excepción producida, normalmente se muestra información adicional (breve descripción, número de línea de código donde falló, volcado de memoria, etc.). La idea de esta información es permitir al usuario y al programador identificar el error. Por ejemplo:

Traceback (most recent call last):

File "<ipython-input-1-05c9758a9c21>", line 1, in <module>

1/0

ZeroDivisionError: division by zero

Podemos observar en qué **fichero** se ha producido el error, en este caso *ipython-input-1-05c9758a9c21* aun cuando no proporciona demasiada información en este caso en particular, muchas veces nos indica el nombre del fichero de código en el cual estamos trabajando (el principal o alguno de los módulos).

También nos indica la **línea** donde se ha producido el error. Aun cuando esta información es, en la mayoría de casos, fiable, también nos podemos encontrar que la línea marca no tenga nada que ver. Con la práctica como programadores aprenderemos a distinguir cuándo este valor es el correcto de cuándo es un valor que no corresponde con la realidad.

En este caso en particular hasta no dice los **valores implicados** en la operación que ha dado la excepción, en este caso ha sido *1/0* 1 dividido por 0, pero no siempre es tan detallado.

Por último, nos indica el **tipo de excepción** *ZeroDivisionError*, y un pequeño **texto descriptivo** *division by zero*.

Para consultar todas las excepciones predefinidas en Python, tenemos la documentación oficial que puede ser consultada en: <https://docs.python.org/3/library/exceptions.html>

En la página web obtendremos todas las excepciones predefinidas en Python y la jerarquía entre ellas. Esta clasificación nos será útil cuando queramos capturarlas, como veremos en el punto siguiente. No es necesario capturarlas una por una, podemos capturar la de nivel superior que engloba a todas sus hijas. Por ejemplo, capturando *BaseException*, capturamos todas o, capturando *LookupError*, capturamos: *IndexError* y *KeyError*. En la figura 43 podemos ver el árbol de excepciones completo.

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        +-- ModuleNotFoundError
    +-- LookupError
        |   +-- IndexError
        |   +-- KeyError
    +-- MemoryError
    +-- NameError
        |   +-- UnboundLocalError
    +-- OSError
        |   +-- BlockingIOError
        |   +-- ChildProcessError
        |   +-- ConnectionError
        |       |   +-- BrokenPipeError
        |       |   +-- ConnectionAbortedError
        |       |   +-- ConnectionRefusedError
        |       |   +-- ConnectionResetError
        |   +-- FileExistsError
        |   +-- FileNotFoundError
        |   +-- InterruptedError
        |   +-- IsADirectoryError
        |   +-- NotADirectoryError
        |   +-- PermissionError
        |   +-- ProcessLookupError
        |   +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
        |   +-- NotImplementedError
        |   +-- RecursionError
    +-- SyntaxError
        |   +-- IndentationError
        |   +-- TabError
    +-- SystemError
    +-- TypeError
    +-- ValueError
        |   +-- UnicodeError
        |       +-- UnicodeDecodeError
        |       +-- UnicodeEncodeError
        |       +-- UnicodeTranslateError
    +-- Warning
        +-- DeprecationWarning
        +-- PendingDeprecationWarning
        +-- RuntimeWarning
        +-- SyntaxWarning
        +-- UserWarning
        +-- FutureWarning
        +-- ImportWarning
        +-- UnicodeWarning
        +-- BytesWarning
        +-- ResourceWarning
    
```

Figura 43. Python built-in exceptions. Fuente: Dominio público.

6.3.3. Gestión de Excepciones

En Python, como en otros lenguajes, podemos delimitar un bloque de código para hacer **captura de las excepciones** que se producen en él e indicar qué **tratamiento** les daremos.

Con la palabra reservada *try* delimitaremos el bloque, y con la palabra clave *except* indicaremos el código para darle tratamiento. Adicionalmente, es importante remarcar que solo daremos tratamiento a las excepciones indicadas en *except*, nombre_excepción, si queremos capturar un conjunto o todas las excepciones deberemos indicar la excepción raíz de ese tipo, consultando la jerarquía en <https://docs.python.org/3/library/exceptions.html> como vimos en el punto anterior. Por ejemplo, en el caso que queramos capturar todas, indicaremos:

try

bloque de código que queremos monitorear las excepciones

except BaseException:

tratamiento para las excepciones

Veamos un ejemplo básico, con un pequeño programa simple donde primero no tratamos las excepciones (ver figura 44), y el mismo programa tratándolas (ver figura 45):

<pre>1 # -*- coding: utf-8 -*- 2 """ 3 @author: Roger 4 """ 5 6 i=1 7 8 while True: 9 10 print(i) 11 i+=1</pre>	<pre>41449 41450 41451 41452 Traceback (most recent call last): File "<ipython-input-3-21cb311eeacc>", line 1, in <module> runfile('C:/Users/Roger/Desktop/Sintitulo1.py', wdir='C:/Users/Roger/Desktop') File "C:\Users\Roger\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 710, in runfile execfile(filename, namespace) File "C:\Users\Roger\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 101, in execfile exec(compile8f.read(), filename, 'exec'), namespace) File "C:\Users\Roger\Desktop\Sin título3.py", line 9, in <module> print(i) File "C:\Users\Roger\Anaconda3\lib\site-packages\ipykernel\iostream.py", line 352, in write self.pub_thread.schedule(lambda: self._buffer.write(string)) File "C:\Users\Roger\Anaconda3\lib\site-packages\ipykernel\iostream.py", line 190, in schedule self.event_pipe.send(event_id)</pre>
---	---

```

File "zmq/backend/cython/socket.pyx", line 636, in
zmq.beckend.cython.Socket.send (zmq\backend\cython\
shocket.c:7305)
File "zmq/backend/cython/socket.pyx", line 683, in
zmq.beckend.cython.socket.Socket.send (zmq\backend\
cython\shocket.c:7048)
File "zmq/backend/cython/socket.pyx", line 201, in
zmq.beckend.cython.socket._send_copy (zmq\backend\c
ython\shocket.c:2920)
File "zmq/backend/cython/checkrc.pxd", line 12, in zmq.
backend.cython.checkrc._check_rc (zmq\backend\cython
\shocket.c:9621)
KeyboardInterrupt

```

Figura 44. Ejemplo de un programa con un bucle infinito, cuando el usuario lo interrumpe con un Ctrl+C se produce una excepción y no se trata. Fuente: Elaboración propia.

<pre> 1 # -*- coding: utf-8 -*- 2 """ 3 @author: Roger 4 """ 5 6 i=1 7 8 while True: 9 try: 10 print(i) 11 i+=1 12 except BaseException as e: 13 print('Excepcion Tratada',e._doc__) 14 break 15 </pre>	<pre> 17396 17397 17398 17399 17400 17401 17402 17403 17404 17405 17406 17407 17408 17409Excepcion Tratada Program interrupted by user. </pre>
---	--

Figura 45. Ejemplo de un programa con un bucle infinito, cuando el usuario lo interrumpe con un Ctrl+C se produce una excepción que en este caso es tratada. Fuente: Elaboración propia.

Como podemos observar, es mucho más elegante tratar las excepciones. En este caso en particular la excepción es una orden para abortar el programa lanzada por teclado pero, en otros casos, la podríamos tratar y recuperarnos para seguir la ejecución.

Veamos cómo es el funcionamiento, que siempre será en este orden:

1. **try** → bloque de código entre **try:** y el **except**
2. **except** → solo si se ha producido una excepción, sino este bloque será saltado.

Siempre se ejecuta el bloque del **try** en primer lugar. Si no hay excepciones, se salta el bloque **except** y se sigue la ejecución normal del código.

Si hay una excepción, en el momento que se produce se salta al *except*. Si el tipo de excepción coincide con el tipo indicado en *except*, se ejecuta el código correspondiente, en el caso de no coincidir la excepción se “lanza” al bloque *try* superior. Si no existe un *try* superior es como si no tuviéramos manejo de excepciones y será el sistema operativo quien decida cómo proceder.

Podemos capturar múltiples excepciones y darles el mismo tratamiento aun si no forman parte de la misma rama, en este caso debemos indicar los diferentes tipos de excepción separados por “,”. En la figura 46 podemos observar la captura de las excepciones *RuntimeError*, *TypeError* y *NameError*, las cuales serán tratadas de la misma forma.

```
except (RuntimeError, TypeError, NameError):
```

Figura 46. Captura de múltiples excepciones: *RuntimeError*, *TypeError* y *NameError*. Fuente: Elaboración propia.

También podemos tratar las diferentes excepciones capturándolas por separado con sendas instrucciones *except* nombre_excepción, indicando para cada tipo el código específico para su tratamiento. En el ejemplo de la figura, indicamos al usuario en castellano cuál ha sido el error. También nos sirve para ilustrar otra funcionalidad como es que un código solo se ejecute si no hay error (excepciones) dentro del bloque *try*. Marcando al mismo nivel que el *try*: con *else*: el código a ejecutar en ese caso. En la figura 47 observamos que en el caso de un simple división nos podemos encontrar con:

- *ValueError*

Que alguno de los números no es entero *ValueError*, excepción que en este caso podría ser lanzada tanto por la conversión a entero *int()* del numerador o del denominador.

- *ZeroDivisionError*

Que el denominador sea igual a 0 *ZeroDivisionError*, como hemos comentado, un ordenador no puede realizar esta operación.

- *Else*

Bloque de código que solo será ejecutado de no producirse un error. Preguntamos al usuario si quiere salir o continuar con otra división.

```
5  @author: Roger
6  """
7
8  while True:
9      try:
10         numerador= int(input('Dame el numerador='))
11         denominador= int(input('Dame el denominador='))
12         print(numerador, '/', denominador, '=', numerador/denominador)
13
14     except ValueError as e:
```

```
15     print('\n Doc=', e.__doc_)
16     print('Los dos numeros deben ser enteros')
17 except ZeroDivisionError as e:
18     print('\n Doc=', e.__doc_)
19     print('Imposible de dividir por 0')
20 else:
21     if('No'== str(input('Otra? (Si o No)'))):
22         break
23
```

Figura 47. Ejemplo con tratamiento diferenciando según la excepción y bloque else para ejecutar en el caso de todo correcto. Fuente: Elaboración propia.

6.3.4. Lanzar una excepción

Como programadores también podemos lanzar (provocar) una excepción usando la palabra reservada `raise` (ver figura 48). Podemos lanzar una de las excepciones predefinidas o podemos definir una propia y lanzarla, en ese caso deberemos hacer que herede de la clase `Exception`, para eso usaremos `class miExcepcion(Exception):` (ver figura 49). No entraremos en más detalles en la parte de las clases, pues el tema se verá en próximas asignaturas sobre Programación Orientada a Objetos.

<pre>1 # -*- coding: utf-8 -*- 2 """ 3 @author: Roger 4 """ 5 6 raise ValueError('Roger')</pre>	<pre>File "C:\User\Roger\Anaconda3\lib\site-packages\spyder\ utils\site\sitecustomize.py",line 101, in execfile exec(compile(f.read(),filename,'exec'),namespace) File "C:/Users/Roger/Desktop/Sin título 1.py",line 6, in <module> raise ValueError('Roger') ValueError: Roger</pre>
---	--

Figura 48. Ejemplo de lanzamiento de una excepción predefinida, en este caso `ValueError`. Fuente: Elaboración propia.

<pre>1 # -*- coding: utf-8 -*- 2 """ 3 @author: Roger 4 """ 5 class miExcepcion(Exception) 6 pass #codigo vacio 7 8 raise miExcepcion('Roger')</pre>	<pre>File "C:\User\Roger\Anaconda3\lib\site-packages\spyder\ut ils\site\sitecustomize.py",line 710, in runfile execfile(filename,namespace) File "C:\User\Roger\Anaconda3\lib\site-packages\spyder\ut ils\site\sitecustomize.py",line 101, in execfile exec(compile(f.read(),filename,'exec'),namespace) File "C:\User\Roger\Desktop\Sin título.py",line 8,in <module> raise miExcepcion('Roger') miExcepcion: Roger</pre>
--	--

Figura 49. Ejemplo de lanzamiento de excepción definida por el programador, en este caso `miExcepcion`. Fuente: Elaboración propia.

6.3.5. Código que siempre se ejecuta

Si queremos que un bloque de código siempre sea ejecutado independientemente de si se produce o no una excepción, podemos usar la palabra reservada *finally* (ver figura 50). Una utilidad del *finally* es, por ejemplo, para cerrar un fichero que tengamos abierto, ya que todo vaya bien o si hay error, de esta forma garantizamos que no queda abierto bloqueando recursos del sistema (ver figura 51).

```
4 @author: Roger
5 """
6
7 while True:
8     try:
9         numerador = int(input('Dame el numerador='))
10        denominador = int(input('Dame el denominador='))
11        print(numerador, '/', denominador, '=', numerador/denominador)
12
13    except ValueError as e:
14        print('\n Doc = ', e.__doc__)
15        print('Los dos numeros deben ser enteros')
16    except ZeroDivisionError as e:
17        print('\n Doc = ', e.__doc__)
18        print('Imposible de dividir por 0')
19    else:
20        if ('No'==str(input('Otra? (Si o No)'))):
21            break
22    finally:
23        print('\n Codigo Finally')
24
```

Figura 50. Ejemplo de la utilización de todas las opciones Try, Except, Else y Finally. Fuente: Elaboración propia.

```
4 @author: Roger
5 """
6
7 try:
8     fichero = open('testRoger.txt', 'w')
9     fichero.write('HOLA')
10    print('Todo OK')
11 except IOError as e:
12    print(e.__doc__)
13 finally:
14    fichero.close()
```

Figura 49. Ejemplo de utilización de Finally para cerrar un fichero y evitar seguir bloqueando recursos del sistema. Fuente: Elaboración propia.

Tema 7.

Mantenimiento del software

El **software** no es algo **estático**, una vez puesto en producción (despliegue, entrega al cliente, etc), entramos en la que será una de las fases más largas de su vida: el **Mantenimiento**.

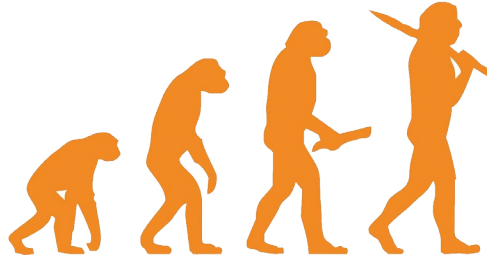
Esta etapa **consume** gran parte del trabajo de los ingenieros de software, normalmente **más que todo el tiempo y esfuerzo** necesarios para poner el software en **producción**. **No solo** incluye la **corrección de errores**, los cuales acostumbran a darse al inicio de la fase de mantenimiento, **también** contempla la **ampliación/actualización de funcionalidades** no prevista inicialmente. Los usuarios muchas veces consideran estas ampliaciones como errores, pues consideran que deberían haber sido incorporadas en la versión inicial.

Según varios estudios, la proporción de trabajo debido a nuevas funcionalidades es aproximadamente del 80% (Eick, S. G. 2001; Pigoski, T. M. 1996).



Software. Fuente: elaboración propia, adaptando imagen CC0 Creative Commons obtenida de pixabay.

Los estudios de Meir M. Lehman et al. de 1969 a 1997 (Lehman, M. M., Meir M., 1980 y Lehman, M. M., Ramil, 1997), lo llevaron a formular lo que se conoce como las **leyes de Lehman**. Estas leyes indican que más que **mantenimiento** del software, lo que sucede es una **evolución** del mismo **en el tiempo**. Todo software, al momento de ponerse en producción, ya está desfasado. Al utilizarse en un **entorno cambiante** (mundo real), o se adapta o irá perdiendo su utilidad hasta que deje de usarse.



Evolución del Software. elaboración propia, adaptando imagen CC0 Creative Commons obtenida de pixabay.

Un **ejemplo extremo** de mantenimiento es el programa de Mecanización de Servicios de Administración de Contratos (MOCA, por sus siglas en inglés) del Departamento de Defensa de los Estados Unidos de América (EE. UU.). El **MOCA** es uno de los softwares más antiguos del planeta que aún sigue **funcionando después de 58 años** de su puesta en marcha. Durante el paso de los años, el sistema se ha **mantenido "actualizado"** porque se le han creado **50 interfaces con programas** financieros más **modernos** para que pudiera seguir cumpliendo su función.

Se **sigue usando** después de tanto tiempo **porque** simplemente **funciona**. Y, adicionalmente, para **reemplazarlo** por un sistema nuevo **costaría**, según cálculos del propio Departamento de Defensa de EE. UU., **US\$142 millones** e implicaría unos **10 años de desarrollo** por 7 grandes empresas especializadas. Su sustitución es un proceso extremadamente complejo.

Para más información pueden consultar el artículo de la BBC " Qué es el MOCA, uno de los softwares más antiguos del planeta que aún sigue funcionando 58 años después de su creación": <http://www.bbc.com/mundo/noticias-41907523>



Sala de la NASA con dos ordenadores IBM 7090. Utilizados para el control de misión de programa espacial Mercury, entorno al 1962, contemporáneos del los equipos del sistema MOCA. Fuente: Wikimedia Commons, NASA, <https://commons.wikimedia.org/wiki/File:NASAComputerRoom7090.NARA.jpg>

7.1. Código heredado

Uno de los principales problemas al realizar mantenimiento de software, es lo que se conoce como código heredado (*legacy code*), software de una cierta antigüedad que tiene asociadas unas problemáticas específicas. Este código heredado forma un volumen muy grande de software que está actualmente en producción y requiere un esfuerzo notable para mantenerlo funcionado y actualizarlo.

Podemos considerar código heredado un software “antiguo” que tiene todas o alguna de las siguientes características:

- Desarrollado hace tiempo.

El suficiente para que nadie recuerde con detalle cómo funciona internamente.

- Usando técnicas y/o herramientas en desuso.

Herramientas no disponibles para el equipo de desarrollo actual o los integrantes del mismo no saben cómo usarlas. Igualmente para las técnicas "sin" nadie familiarizado en las utilizadas en su momento para el desarrollo.

- Por personas que ya no pertenecen a la organización.

En muchas ocasiones se ve agravado porque las personas directamente ya no trabajan en la organización y no pueden ni refrescar los conocimientos para volver a trabajar en el software o hacer el traspaso del conocimiento al nuevo equipo encargado.

- El cual ha sufrido diversos mantenimientos.

El software ha sufrido diversos mantenimientos, en muchas ocasiones documentados pobremente o sin ningún tipo de documentación que aún hacen más difícil la comprensión.

- Tirarlo a la basura y empezar de nuevo no es opción.

Aun con todos los problemas nombrados hasta ahora, uno pensaría que la mejor medida es directamente descartarlo y empezar de nuevo, pero muchas veces no es posible por:

- Coste:

La inversión realizada inicialmente en el código y/o la inversión necesaria para realizar un nuevo desarrollo desde 0.

- Tiempo:

El tiempo necesario para realizar el desarrollo desde 0 es demasiado elevado y no podemos detener la organización mientras lo realizamos. Si queremos realizar en paralelo el mantenimiento del día a día para que siga funcionando junto con el desarrollo nuevo, a veces el coste de mantener los dos equipos de trabajo no será asumible.

- Amortización:

Simplemente el coste del software fue muy elevado y la organización no está dispuesta a dar ese dinero por perdido.

- Equipos hardware antiguos:

Muchas veces la interdependencia entre el hardware y el software antiguo es tal que no permite cambiar uno sin cambiar el otro. Llevando por ejemplo a la necesidad de actualizar un hardware de alto coste que no permite un nuevo desarrollo con herramientas y lenguajes modernos porque simplemente es incompatible. La opción sería ir adaptando el software para que siga funcionando con el hardware original.



Legacy code. Fuente: imagen obtenida de pixabay, CC0 Creative Commons.

Las dificultades específicas que debemos abordar en el mantenimiento del código heredado forman parte de las leyes de Lehman. Las cuales veremos con detalle en el próximo punto.

7.2. Leyes de Lehman

Como comentamos anteriormente, los estudios de Meir M. Lehman et al., de 1969 a 1997 (Lehman, M. M., Meir M., 1980 y Lehman, M. M., Ramil, 1997) establecieron lo que se conoce como leyes de Lehman. Las leyes fueron formuladas en diferentes épocas.

7.2.1. Continuidad del Cambio

Formulada en 1974.

El software, como se comentó al principio del tema, no es estático, siempre está en evolución al encontrarse funcionando en el mundo real. Los motivos de esta evolución se pueden resumir en:

- **Se detecta la necesidad de nuevas funcionalidades cuando comienzan a utilizar el software.**

Los usuarios, cuando indican lo que quieren que realice el software, siempre se dejan detalles que, posteriormente, les llevarán a pedir estas funcionalidades "olvidadas" o, simplemente al

ver las posibilidades que el nuevo software les provee, encuentran nuevas funcionalidades que antes simplemente no habían pedido por desconocer que eran factibles.

- **Mejoras en el hardware permiten mejoras en el software.**

El hardware siempre está en evolución, cada año mejoran sus capacidades de forma notable. Hay funcionalidades o requisitos que en el momento de concepción del software no eran posibles por su complejidad computacional (por ejemplo, el tiempo de cómputo era excesivo haciendo los datos obtenidos obsoletos antes de obtenerlos) o por la falta de algún dispositivo (por ejemplo, cuando salieron los lectores de *Radio Frequency IDentification* (RFID), el control de inventario pudo hacerse de mejor manera usando esta nueva tecnología).

- **Se detectan errores.**

Simplemente en la verificación no se detectó un error y hay que corregirlo.

- **Cambio de Sistema Operativo (SO) o Hardware, el software debe reinstalarse.**

El sistema operativo cambia de versión o deja de estar mantenido por el fabricante y es necesario cambiarlo. El Hardware puede dañarse y no encontrar repuestos o a un coste exorbitante que aconsejen cambiar la plataforma. En ambos casos, deberemos reinstalar el software y, a veces, eso implicará modificar el código para que pueda seguir funcionando. Por ejemplo, Java se supone que es portable, pero puede ser que la versión deje de ser compatible porque alguna funcionalidad interna que usamos deje de estar implementada en la nueva versión.

- **Mejorar eficiencia.**

El software estaba pensado para un volumen de usuarios y datos determinados y estos, con el tiempo, van aumentando, por ello es necesario mejorar la eficiencia, para poder seguir dando las funcionalidades requeridas en los tiempos de respuesta adecuados. También, si nuestro software dependía de otro software con el cual se comunicaba para completar alguna funcionalidad, si el otro software cambia, el nuestro debe también adaptarse al cambio.

7.2.2. Incremento de la Complejidad

Formulada en 1974.

En cada nueva modificación que se realiza en un software, lleva casi siempre aparejada un aumento de su complejidad.

Tratar de evitar que la complejidad aumente requiere de un esfuerzo adicional considerable por parte del equipo de desarrollo, con un coste monetario mayor y la dedicación de tiempo adicional al necesario para implementar la modificación.

Aun así, de contar con todos los recursos necesarios, puede ser inevitable el aumento de complejidad, por ejemplo al incorporar nuevas funcionalidades también estamos aumentando los posibles puntos de error.

7.2.3. Evolución del Programa

Formulada en 1974.

La evolución de los programas es un proceso autorregulado. El contexto donde este se encuentra determina cómo será su evolución y se pueden observar tendencias invariantes como, por ejemplo, el tiempo entre versiones (cada nueva evolución programada), la cantidad de errores que se le detectan (el equipo y las metodologías no acostumbran a cambiar y producen resultados similares de calidad) o el tamaño (cómo evoluciona la cantidad/complejidad en cada nueva versión).

7.2.4. Conservación de la Estabilidad Organizacional

Formulada en 1980.

En la vida de un programa, la dedicación que requiere es aproximadamente constante e independiente de los recursos dedicados. Relacionada con la anterior de la autorregulación y, quizá una de las más "polémicas" pues establece que es difícil que la velocidad de desarrollo de un programa aumente porque, según Lehman, casi nunca se tendrán las condiciones ideales para ello: o el equipo de trabajo no será competente, o los directores del desarrollo no sabrán por dónde ir, o los directivos aplazarán el proyecto, o se tomarán decisiones erróneas, o se deberá reducir el presupuesto (menos equipo humano o menos recursos técnicos). En definitiva, siempre habrá un motivo para que la velocidad de desarrollo se mantenga o disminuya, pero nunca aumente.

7.2.5. Conservación de la Familiaridad

Formulada en 1980.

En la vida de un programa, la cantidad de cambios que trae cada nueva versión del mismo (*release*) es constante (aproximadamente).

Programadores, comerciales y usuarios deben mantener la familiaridad con el programa para que las evoluciones sean aceptadas por todos ellos.

Si el sistema evoluciona demasiado rápido o demasiado lento, alguno de los actores implicados presionará para que se estabilice.

7.2.6. Crecimiento continuo

Formulada en 1980.

Para que los usuarios sigan satisfechos con el software, este debe ir incorporando progresivamente nuevas funcionalidades.

7.2.7. Declive de la calidad

Formulada en 1996.

Con el incremento de la complejidad, la calidad del software bajará a no ser que se dedique una gran cantidad de recursos para evitarlo. Relacionada directamente con la ley **Incremento de la Complejidad (1974)**.

7.2.8. Retroalimentación del sistema

Formulada en 1996.

El mantenimiento del software es un proceso que se retroalimenta en cada nueva iteración de mantenimiento, considerando los diferentes actores y niveles de la organización implicados. Deben considerarse todos los factores para que el mantenimiento pueda ser realizado de la forma más satisfactoria posible.

7.3. Problemas generales

Adicionalmente a lo indicado por Lehman, hay otros problemas recurrentes en el mantenimiento del software. El mantenimiento no puede ser realizado sobre la marcha y *ad hoc* al problema puntual, hace falta realizar una correcta gestión para poder **minimizar** todos los **problemas** nombrados en las diferentes leyes.

Se necesita dedicar los **recursos adecuados** para tener un **equipo** o dar a los programadores de desarrollo la descarga necesaria para poder abordar adecuadamente la tarea, usando una **metodología** que permita dividirla en actividades concretas y establezca los criterios de calidad y supervisión adecuados para que el mantenimiento degrade lo mínimo posible el software, así como **documentar los cambios** de forma clara.

Hay un factor que muchas veces se olvida y que son los **usuarios**, los cuales deben formar parte de la validación del mantenimiento realizado para garantizar que el software sigue cumpliendo lo que se espera de él y escuchar las sugerencias de funcionalidad que propongan. Un **software sin usuarios** es un software que no se usará y **acabará** siendo **descartado**.

7.4. En el Grado Ingeniería Informática

La mayoría de metodologías de desarrollo de software que se verán durante el grado (en futuras asignaturas) contemplan la fase de mantenimiento. También, se darán técnicas (de Ingeniería de Software) para procurar un mantenimiento efectivo y eficiente.

Ahora, en Metodología de Programación, nos concentraremos en hacer las cosas bien desde el principio, **estructurando** y **documentando** nuestros programas.



Una buena estructura es parte fundamental de un buen programa. Fuente: imagen obtenida de pixabay, CC0 Creative Commons.



Un código bien documentado, es más fácil de corregir, mantener y evolucionar. Fuente: imagen obtenida de pixabay, CC0 Creative Commons.

Para **estructurar**, usaremos lo que hemos visto de dividir los códigos complejos en **funciones**. Si tenemos un grupo de funciones considerable que aborda un mismo problema, las agruparemos en **módulos**. Si tenemos muchos módulos que están relacionados entre ellos, los agruparemos para formar un **paquete**. Lo haremos siguiendo las pautas que hemos visto anteriormente tanto para crear módulos como para crear paquetes.

En cuanto a la **documentación**, deberemos indicar para cada **bloque** de código del programa, **función**, **módulo** y **paquete**, una **breve descripción y cómo usarlo**. Si tenemos partes del programa que son enrevesadas o no son intuitivas, también deberemos poner un pequeño comentario indicando qué estamos haciendo en el código y para qué lo estamos haciendo. Una correcta **documentación** nos servirá para poder **entender** nosotros mismo el **código** cuando lo revisemos después de un tiempo y, lo más importante, permitirá que otros lo entiendan, le puedan hacer mantenimiento y lo reutilicen y usen correctamente.

Tema 8.

E/S (Input/Output), ficheros

Los ficheros permiten almacenar gran cantidad de información de forma permanente. En este tema veremos cómo usarlos en nuestros programas.

En la documentación podemos encontrar diferentes formas de referirse a lo mismo:

- E/S.
- Entra / Salida.
- I/O.
- Input / Output.

Las cuatro formas se refieren a los mismo, a las operaciones mediante las cuales damos datos para ser tratados a un programa o cómo este nos comunica los resultados.

8.1. Entra/Salida

Para interactuar con los programas y realizar operaciones de entrada y/o salida de datos hemos visto hasta ahora ejemplos por **pantalla** para mostrar los datos y, por **teclado**, para introducirlos. Usando `input()` y `print()` en nuestro caso particular del lenguaje Python.

Cuando queremos administrar un **volumen** más **grande de datos** o queremos que nuestros **datos persistan** más allá de la ejecución del programa, estos dos sistemas nos limitan y estas formas básicas ya no nos sirven. Ya sea por volumen o por persistencia, los **ficheros** son una buena alternativa. Hay otras como, por ejemplo, las Bases de Datos, que se verán más adelante en el grado en las asignaturas específicas.

8.2. Ficheros en Python

El primer paso para trabajar con ficheros es abrirlos usando la función `open()`, en la cual indicaremos el nombre del fichero y en qué modo lo queremos abrir:

`open(filename, mode)`

Una vez abierto correctamente y según el modo en que lo hayamos hecho, disponemos de varias funciones para trabajar:

`file.read(size)`

`file.readline()`

`file.readlines()`

`list(f)`

`file.write(string)`

`f.tell()`

`f.seek(offset, from_what)`

file.close()

Vamos a ver con detalle cada una de ellas.

8.2.1. **open()**

file open(nombre_fichero, modo_apertura)

open() abre el fichero con nombre igual a *nombre_fichero*, en el modo de apertura indicado por *modo_apertura*, regresa un objeto del tipo *fichero*. Según el modo elegido podemos tener error si no existe o podemos provocar la rescritura del fichero.

- **Modos**

'r'

Solo lectura. Si el fichero no existe lanza una excepción del tipo *FileNotFoundError*.

'w'

Crea el fichero y lo abre de solo escritura, si existe un fichero con el nombre indicado elimina su contenido (lo reescribe). El fichero no puede estar abierto por otro programa o tendremos conflicto.

'x'

Crea el fichero y lo abre de solo escritura, si existe un fichero con el nombre indicado lanza una excepción del tipo *FileExistsError*.

'a'

Abre en modo escritura, pero añadiendo al final (*append*) en el caso que el fichero exista. Si el fichero no existe lo crea. El fichero no puede estar abierto por otro programa o tendremos conflicto.

'r+'

Abre el fichero en modo lectura y escritura. Si el fichero no existe lanza una excepción del tipo *FileNotFoundError*.

't'

Trabaja con el fichero en modo texto. Las operaciones sobre el fichero de lectura o escritura deberán hacerse con *strings (str)*.

'b'

Trabaja con el fichero en modo binario. Las operaciones sobre el fichero de lectura o escritura deberán hacerse con *bytes* sin ningún tipo de codificación o decodificación.

Por defecto, si no indicamos nada, se abre en modo 'r' y en modo texto 't'. Para combinar diferentes opciones utilizaremos ambas letras, por ejemplo:

`'rt'`

Modo lectura y texto. Es la opción por defecto si no indicamos nada.

`'wb'`

Modo escritura y en binario.

En el proyecto de la asignatura solo nos centraremos en el modo texto, utilizaremos los diferentes modos de apertura disponibles según nuestra necesidad.

Hay más parámetros que no veremos por ahora y en los cuales utilizaremos los valores por defecto:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True,
opener=None)
```

Si alguien tiene curiosidad, puede consultar la documentación oficial de Python donde se explican todos los parámetros en detalle: <https://docs.python.org/3/library/functions.html#open>

A continuación, dos ejemplos. En el primero de ellos vemos qué pasa cuando abrimos un fichero en un modo incorrecto para lo que queremos hacer, en este caso abrimos sin indicar el modo (se intentara abrir con modo 'rt') y, como el fichero no existe, se lanza una excepción (ver figura 52).

En el segundo ejemplo abrimos el fichero en modo escritura 'w', en este caso se crea y nos deja escribir el texto 'HOLA' (ver figura 53).

<pre> 10 try: 11 f=open('test.txt') 12 f.write('HOLA') 13 f.close() 14 15 except OSError: 16 print('Error Open:',OSError.__doc__) 17 </pre>	<pre> In [3]: runfile('C:/Users/Roger/.spyder- py3/temp.py', Error Open: Base class for I/O related errors. In [4]: </pre>
---	---

Figura 52. Ejemplo Open() en modo incorrecto para hacer una escritura. Fuente: Elaboración propia.



Figura 53. Ejemplo Open() en modo correcto para hacer una escritura. Fuente: Elaboración propia.

8.2.2. read()

`file.read(size)`

Lee del fichero la cantidad de información indicada por `size`. El parámetro tamaño (`size`) es opcional, si no indicamos el tamaño, leerá todo el fichero. No es recomendable **leer todo el fichero** de golpe, pues podemos tener **problemas de memoria** si el tamaño del mismo es grande.

Es importante señalar que la función `read()` solo puede ser invocada desde un objeto del tipo fichero. Primero abriremos el fichero `f=open()` y sobre el objeto retornado invocaremos el `read()`: `f.read()`.

Para poder leer por bloques, los ficheros tienen un cursor/apuntador que guarda en qué posición estamos, como por ejemplo se hace en un editor de textos. Las operaciones de lectura y escritura se hacen tomando como referencia este cursor.

En los siguientes tres ejemplos vemos qué pasa cuando no indicamos tamaño (`size`) o si indicamos uno más pequeño que el contenido del fichero. El fichero `test.txt` solo contiene el texto 'HOLA'. En un primer caso realizamos un `read()` sin indicar el tamaño a leer (ver figura 54), en el segundo ejemplo hacemos una lectura con tamaño igual a 1 (ver figura 55), y en el último ejemplo hacemos la lectura con tamaño igual a 3 (ver figura 56).

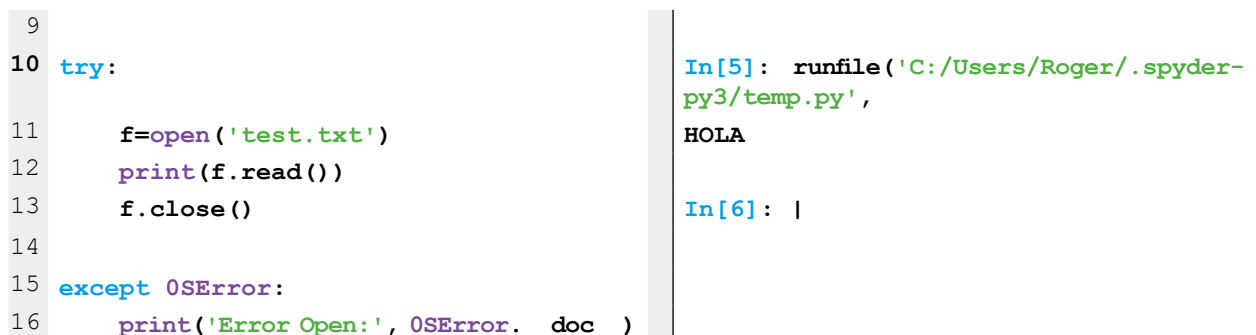


Figura 54. Ejemplo Read() sin indicar tamaño. Fuente: Elaboración propia.

```

9
10 try:
11     f=open('test.txt')
12     print(f.read(1))
13     f.close()
14
15 except OSError:
16     print('Error Open:', OSError.__doc__)

```

```

In[6]: runfile('C:/Users/Roger/.spyder-
py3/temp.py')
H

In[7]:

```

Figura 55. Ejemplo Read() con tamaño igual a 1. Fuente: Elaboración propia.

```

9
10 try:
11     f=open('test.txt')
12     print(f.read(3))
13     f.close()
14
15 except OSError:
16     print('Error Open:', OSError.__doc__)

```

```

In[7]: runfile('C:/Users/Roger/.spyder-
py3/temp.py')
HOL

In[8]: |

```

Figura 56. Ejemplo Read() con tamaño igual a 3. Fuente: Elaboración propia.

Aun cuando son ejemplos consecutivos, en cada ocasión el cursor del fichero se sitúa siempre en la primera posición con el open(), no conserva la posición de la anterior ejecución.

En el caso que tuviéramos abierto el fichero para lectura y escritura, si realizásemos una lectura y después una escritura, esta se realizaría al final. Para el mismo fichero anterior podríamos intentar insertar un '8' después de la segunda posición (ver figura 57) sin éxito.

```

7
8 try:
9     f=open('test.txt', 'r+')
10    print(f.read(2))
11    f.write('8')
12    f.close()
13    f.= open('test.txt')
14    print(f.read())
15    f.close()
16
17 except OSError:
18    print('Error:',OSError.__doc__)
19    f.close()

```

```

In[27]: runfile('C:/Users/Roger/.Desktop')
HO
HOLA8

In[28]:

```

Figura 57. Ejemplo Read() con tamaño igual a 2, seguido de una escritura. Fuente: Elaboración propia.

8.2.3. readline()

`file.readline()`

Lee una línea del fichero a la vez. Añade '\n' al final de cada línea leída, que es el símbolo para un salto de línea.

Cuando llegamos al final del fichero regresa "", *string* vacío. En los dos ejemplos siguientes, el fichero test.txt tiene como contenido 'HOLA1\nHOLA2\nHOLA3\nHOLA4' (ver figura 58).

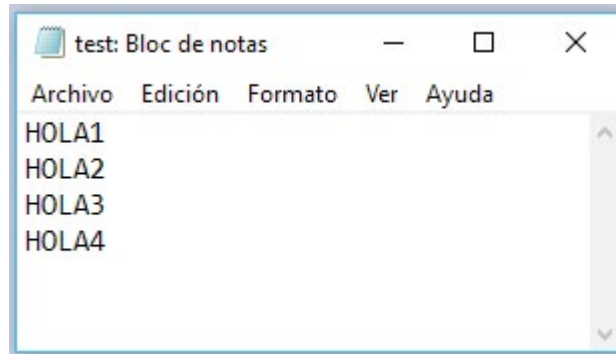


Figura 58. Fichero de texto con cuatro HOLAS, uno por línea. Fuente: Elaboración propia.

En el primer ejemplo, leamos una sola línea utilizando `readline()` (ver figura 59). En el segundo, realizamos diversas llamadas consecutivas a `readline()` y observamos que, de forma automática, el cursor cada vez está en la siguiente línea y que nos introduce un salto de línea entre cada lectura (ver figura 60).

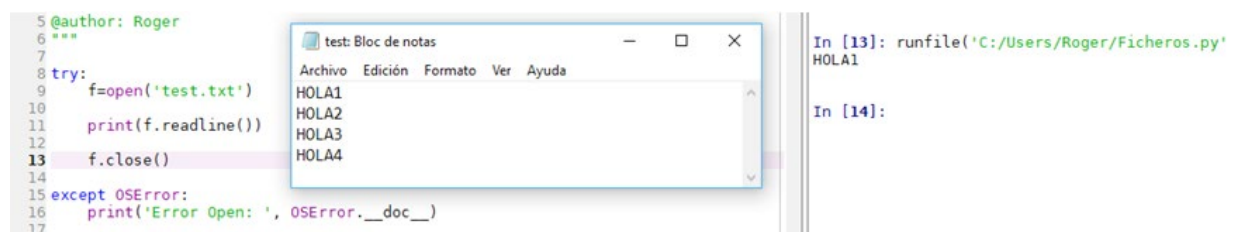


Figura 59. Ejemplo Readline(), una sola llamada. Fuente: Elaboración propia.

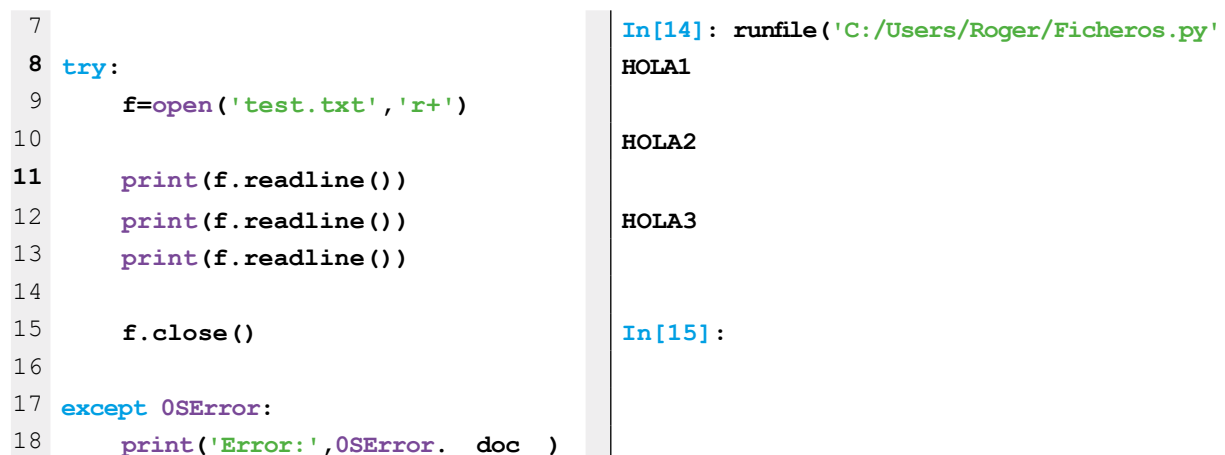


Figura 60. Ejemplo Readline(), llamadas consecutivas, a cada llamada el cursor está en la siguiente línea de forma automática. Fuente: Elaboración propia.

Si realizamos 5 lecturas con `readline()`, la última de ellas nos regresará un str vacío, tenemos solo 4 HOLAS (ver figura 61).

```
7
8 try:
9     f=open('test.txt')
10    print('1 readline=',f.readline())
11    print('2 readline=',f.readline())
12    print('3 readline=',f.readline())
13    print('4 readline=',f.readline())
14    print('5 readline=',f.readline())
15    f.close()
16
17 except OSError:
18    print('Error:',OSError.__doc__)
19    f.close()
20
```

```
In[28]: runfile('C:/Users/Roger/Desktop/Sin
título 1.py',wdir='C:/Users/Roger/Desktop')
1 readline = HOLA1
2 readline = HOLA2
3 readline = HOLA2
4 readline = HOLA4
5 readline =

In [29]:
```

Figura 61. Ejemplo `Readline()`, 5 llamadas consecutivas hasta leer todo el fichero. Fuente: Elaboración propia.

8.2.4. `readlines()` / `list()`

`file.readlines()`

`list(f)`

Ambas funciones leen todas las líneas de un archivo y las guardan en un vector (*list*).

Si lo único que queremos es leer línea por línea y no necesitamos guardar para más cálculos o transformaciones la misma, hay un manera más eficiente de leerlas utilizando *for line in f*.

En los tres ejemplos para ilustrar ambas opciones y para el *for* se utiliza el mismo fichero `test.txt` del apartado anterior con el mismo contenido `'HOLA1\nHOLA2\nHOLA3\nHOLA4'` (ver figura 58). En el primer ejemplo, simplemente hacemos un `readlines()`, como se guarda en un vector, la salida por pantalla del `print()` está delimitada por `[]` (ver figura 62). En el segundo ejemplo, obtenemos el mismo resultado utilizando `list()` (ver figura 63). En el último ejemplo, se muestra una manera más eficiente de leer y tratar cada línea de forma individual (ver figura 64).

```
8 try:
9     f=open('test.txt')
10
11    print(f.readlines())
12
13    f.close()
14
15 except OSError:
16    print('Error Open:', OSError.__doc__)
```

```
HOLA3

In [15]: runfile('C:/Users/Roger/Ficheros.py')
['HOLA1\n', 'HOLA2\n', 'HOLA3\n', 'HOLA4\n']

In [16]
```

Figura 62. Ejemplo `Readlines()`. Se guarda en un vector `['HOLA1\n', 'HOLA2\n', 'HOLA3\n', 'HOLA4']`. Fuente: Elaboración propia.

```
7
8 try:
9     f=open('test.txt')
10
11     print(list(f))
12
13     f.close()
14
15 except OSError:
16     print('Error Open:', OSError.__doc__)
```

```
In [17]: runfile('C:/Users/Roger/Ficheros.py')
['HOLA1\n', 'HOLA2\n', 'HOLA3\n', 'HOLA4']

In [18]:
```

Figura 63. Ejemplo List(). Se guarda en un vector = ['HOLA1\n', 'HOLA2\n', 'HOLA3\n', 'HOLA4']. Fuente: Elaboración propia

```
5 @author: Roger
6 """
7
8 try:
9     f=open('test.txt')
10
11     for line in f:
12         print(line)
13
14     f.close()
15
16 except OSError:
17     print('Error Open:', OSError.__doc__)
```

```
In[16]: runfile('C:/Users/Roger/Ficheros.py')
HOLA1
HOLA2
HOLA3
HOLA4

In[17]:
```

Figura 64. Ejemplo for line in f). Se trata cada línea de forma individual, en este caso solo se imprime por pantalla pero se podría realizar un procesamiento más complejo. Fuente: Elaboración propia.

8.2.5. write()

`file.write(string)`

Escribe el *string* en el archivo, regresa el número de caracteres escritos. En el ejemplo, podemos ver cómo creamos un archivo (o lo sobrescribimos si ya existe) llamado test.txt donde ponemos la palabra 'Chao' de 4 letras/posiciones. En el ejemplo también vemos que la función regresa la cantidad de letras escritas, en este ejemplo 4. En la segunda parte del código del ejemplo, simplemente lo cerramos y abrimos en modo lectura para comprobar que la operación de escritura se ha realizado correctamente (ver figura 65).

```
7
8 try:
9     f=open('test.txt', 'w')
10
11     numChar=f.write('Chao')
12
```

```
In[42]: runfile('C:/Users/Roger/Ficheros.py',
numChar = 4
Leer primera linea = Chao

In [43]:
```

```
13     print('numChar=', numChar)
14
15     f.close()
16
17     f=open('test.txt', 'r')
18
19     print('Leer primera linea=', f.
20         readline())
21
22     f.close()
23 except OSError:
24     print('Error Open', OSError.__doc__)
25
```

Figura 65. Ejemplo write(). Fuente: Elaboración propia.

Recordemos que, por defecto, los ficheros se abren en modo texto, si el fichero hubiera sido abierto en modo binario no podríamos realizar la operación de escritura con un *str*. En ese caso, el programa lanzaría una excepción *TypeError* (ver figura 66).

```
6  """
7
8  try:
9      f=open('test2.txt', 'wb')
10
11      f.write('HOLA')
12
13      f.close()
14
15  except OSError:
16      print('Error:', OSError.__doc__)
17      f.close()
18
```

```
File "C:\Users\Roger\Anaconda3\lib\site-packa
ges\spyder\utils\site\sitecustomize.py",
line 710, in runfile
    execfile(filename, namespace)

File "C:\Users\Roger\Anaconda3\lib\site-pack
ages\spyder\utils\site\sitecustomize.py",
line 101, in execfile
    exec(compile(f.read(), filename,
'exec'), namespace)

File "C:\Users\Roger\Desktop\Sin título1.py",
line 11, in <module>
    f.write('HOLA')
TypeError: a bytes-like object is
required, not 'str'
```

Figura 66. Ejemplo write() de un string en un fichero abierto en modo binario. Fuente: Elaboración propia.

8.2.6. tell() / seek()

f.tell()

f.seek(offset, from_what)

- **f.tell()**

Devuelve la posición actual del cursor. En ficheros binarios la posición es en bytes, en ficheros de texto es un número indeterminado.

- **f.seek(offset, from_what)**

Modifica la posición del cursor. *offset* indica la cantidad de posiciones a desplazarse. *from_what* indica el punto de referencia:

- 0 = des del inicio del fichero.
- 1 = posición actual del cursor.
- 2 = des del final del fichero.

En los ficheros de texto solo podemos usar *seek(x, 0)* donde x puede ser cualquier *offset* o *seek(0, 2)* para situarnos al final. No es posible utilizar la posición actual del cursor como referencia, pues el número devuelto como posición es un número indeterminado.

En el ejemplo podemos ver cómo, al abrir el fichero, el cursor se encuentra en la primera posición, la 0. Después de leer un carácter, se encuentra en la posición 1. Y si usamos *seek()* para posicionarnos en la posición 4 desde el inicio del fichero, *tell()* nos indica que estamos en esa posición (ver figura 67).

```
8 try:
9     f=open('test.txt','w')
10    numChar=f.write('1234567890')
11    print('numChar=',numChar)
12    f.close()
13
14    f=open('test.txt','r')
15    print('Al abrir el fichero tell()=',f.
16          tell())
17    print('Read(1)',f.tell(1))
18    print('Al leer 1 caracter tell ()=',
19          f.tell())
20    f.seek(4, 0)
21    print('Al leer despues de
22          posicionar tell()=',f.tell())
23    f.close()
24
25 except OSError:
26     print('Error Open:',OSError.__doc__)
```

```
In[54]: runfile('C:/Users/Roger/Ficheros.py',
numChar = 10
Al abrir el fichero tell() = 0
Read(1) = 1
Al leer 1 caracter tell() = 1
Al leer despues de posicionar tell ()=4
```

```
In[55]:
```

Figura 67. Ejemplos con *tell()* para saber la posición actual y con *seek()* para mover el cursor. Fuente: Elaboración propia.

8.2.7. close()

file.close()

Cierra el fichero y libera los recursos asociados.

Es importante cerrar los ficheros, aun cuando Python lo hace después de un tiempo prudencial de forma automática. Mantener los ficheros abiertos consume recursos del sistema, es recomendable cerrarlos tan pronto como sea posible, sin abusar, pues si abrimos y cerramos continuamente también consumimos recursos de forma ineficiente. Por norma general debemos cerrar el fichero cuando no lo volvamos a necesitar (leer o escribir en él) por un tiempo indeterminado o si ya sabemos que no lo utilizaremos más.

Tema 9.

Proyecto informático de programación

La asignatura tiene un **fuerte componente práctico**, cada semestre se desarrollarán los conceptos teóricos de este manual mediante un **proyecto**. El proyecto se realizará de forma **incremental**, añadiendo los conceptos teóricos de forma paulatina a medida que se vayan viendo los temas correspondientes en clase.

Nos basaremos en el modelo en espiral desarrollado por Boehm (Boehm, B. W., 1988) (ver figura 68).

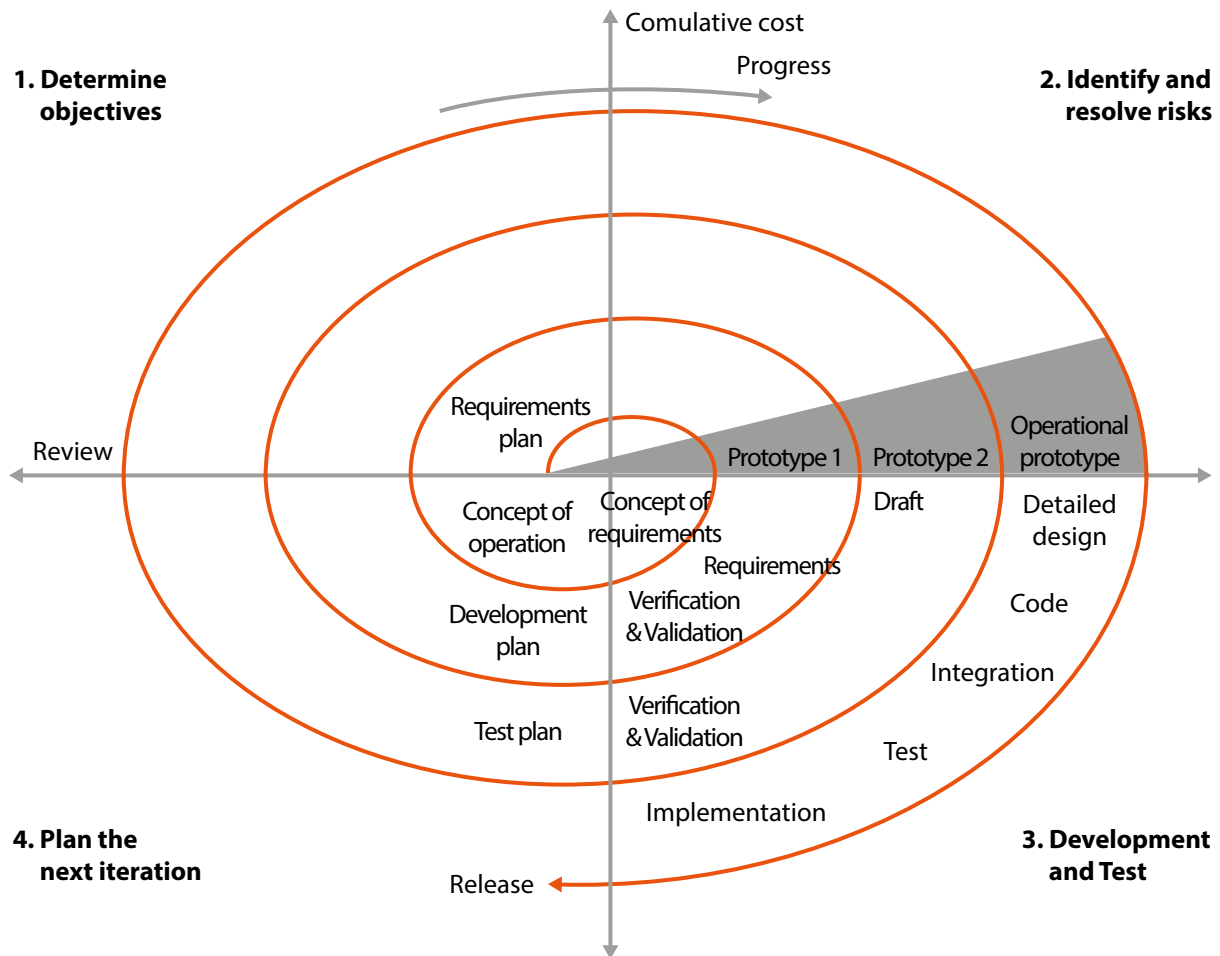


Figura 68. Resumen gráfico del modelo en espiral de Boehm. Fuente: Public Domain

9.1. Ejemplo de un proyecto

Tendremos una temática que se irá desarrollando durante todo el curso. Para este ejemplo:

- Vamos a desarrollar un software para control de un **complejo logístico de hidrocarburos**.
- El proyecto será individual (según el curso, también puede darse el caso de que sea en grupo), pero se intercambiará código entre los diferentes estudiantes del curso para comprobar la correcta "modulación" de los 2 componentes.
- El proyecto se irá construyendo de forma incremental durante toda la asignatura.



Unidad de bombeo en una perforación petrolera. Fuente: imagen bajada de pixabay, CC0 Creative Commons.

El proyecto será dividido en diferentes componentes que se irán entregando durante el curso para su verificación. Puede ser que algún componente, debido a su complejidad, sea elaborado en dos entregas. La entrega final será la integración de todos los componentes para que trabajen como un solo programa.

9.1.1. Componente 1: Gasolinera

Crear un programa que simule una gasolinera.

Debe soportar diferentes tipos de combustibles, atender a los clientes y recibir reaprovisionamiento de combustible.

Funciones a implementar:

`bool configInicial(diccionario_combustible_cantidad_capacidad).`

`bool ponerCombustible(combustible, cantidad).`

`bool recibirSuministro(diccionario_combustible_cantidad).`

Para cada función, se indicará una descripción más detallada de cuál debe ser su comportamiento.



Surtidor de combustible. Fuente: imagen bajada de pixabay, CC0 Creative Commons.

bool configInicial(diccionario_combustible_cantidad_capacidad)

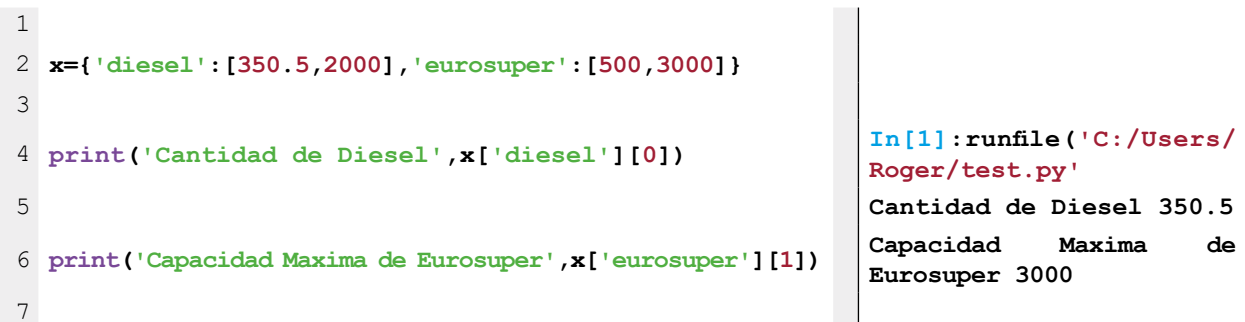
Configuración Inicial de la Gasolinera. Debe retornar *True* si todo va bien y *False* si hay algún tipo de error (por ejemplo, cantidad o capacidad negativos o *strings*, cantidad mayor a capacidad, etc.).

Inicializará los tipos de combustible que dispone la gasolinera con la capacidad máxima del tanque de cada tipo de combustible y su stock/nivel actual.

Ejemplo:

```
True ← configInicial({'diesel':[350.5,2000], 'eurosuper':[500,2000]})
```

Es posible que el valor de un diccionario sea a su tiempo una lista con dos valores (ver figura 69).



```

1
2 x={'diesel':[350.5,2000], 'eurosuper':[500,3000]}
3
4 print('Cantidad de Diesel',x['diesel'][0])
5
6 print('Capacidad Maxima de Eurosuper',x['eurosuper'][1])
7

```

```

In[1]: runfile('C:/Users/Roger/test.py')
Cantidad de Diesel 350.5
Capacidad Maxima de Eurosuper 3000

```

Figura 60. Ejemplo de diccionario. Fuente: Elaboración propia.

bool ponerCombustible(combustible, cantidad)

Un cliente quiere poner la cantidad indicada del combustible indicado.

Retorna *True* si todo esta OK, *False* en caso de algún error con los parámetros (valores incorrectos).

Si hay suficiente cantidad, debe servir al cliente y restar la cantidad servida al stock. Si no hay suficiente stock, debe indicar cuántos litros se pudieron servir. Si la cantidad de combustible del tipo requerido es 0, debe indicarse 'Agotado'.

Si el tipo de combustible no existe en la gasolinera, debe indicar 'Combustible no disponible en esta estación'.

```
ponerCombustible('diesel', 15.5)
```

bool recibirSuministro(diccionario_combustible_cantidad)

La gasolinera recibe reaprovisionamiento, puede recibir varios tipos de combustible a la vez (por eso se usa un diccionario).

Para cada tipo de combustible suministrado se debe actualizar el stock.

Retorna *True* si todo esta OK, *False* en caso de algún error con los parámetros (valores incorrectos).

En el caso de que la gasolinera no tenga alguno de los tipos de combustible mandados con las remesas, debe dar un mensaje de alerta y seguir con el siguiente tipo.

En el caso de que el suministro más el stock actual superen la capacidad, debe llenarse hasta la capacidad máxima y notificar que el tanque está lleno al máximo.

Ejemplo:

```
True ← recibirSuministro ({'diesel':750, 'eurosuper':800})
```

```
def recibirSuministro(s)
```

```
...
```

```
if x['diesel'][0]+s['diesel'] <= x['diesel'][1]:
```

```
    x['diesel'][0] = x['diesel'][0] + s['diesel']
```

```
else:
```

```
    x['diesel'][0] = x['diesel'][1]
```

9.1.2. Componente 2: Refinería

Se desarrollará un programa que simule el comportamiento de una refinería. Es una aproximación al proceso, no pretende ser realista.

La refinería recibe envíos de petróleo, alcohol y aceite vegetal y produce gasolina95, gasolina98, diésel y biodiésel.

Trabajamos siempre en litros, de cada litro de petróleo se obtiene:

- 14,8% de componente base gasolina.
- 16,1% de diésel refinado.
- El resto son derivados que no nos interesan para este proyecto.

Para poder “producir” los diferentes combustibles se utiliza:

- gasolina95 = 95% componente base gasolina + 5% alcohol.
- gasolina98 = 90% componente base gasolina + 10% alcohol.
- diésel = 100% diésel refinado.
- biodiésel = 50% diésel refinado + 50% aceite vegetal.

Funciones a implementar:

`bool configInicial(diccionario_materias_primas_y_combustibles_cantidad_capacidad).`

`diccionario_real_combustible_cantidad pedirSuministro(diccionario_combustible_cantidad).`

`int refinar(combustible, litros).`

`infoMediaDias(dias).`

`bool recibirMateriaPrima(diccionario_materia_cantidad).`



Refinería. Fuente: imagen bajada de pixabay, CC0 Creative Commons.

`bool configInicial(diccionario_materias_primas_y_combustibles_cantidad_capacidad)`

Configuración Inicial de la Refinería.

Debe retornar *True* si todo va bien y *False* si hay algún tipo de error.

El diccionario tendrá los datos de cantidad actual y capacidad máxima de: petróleo, alcohol, aceite vegetal, gasolina95, gasolina98, diésel y biodiésel.

{'producto':[cantidad, max],...}

Ejemplo:

{'petroleo': [500, 20000], 'alcohol': [200, 1000],}

`diccionario_real_combustible_cantidad pedirSuministro(diccionario_combustible_cantidad)`

Hacer la solicitud a la refinería para un suministro de combustibles.

La refinería tratará de servir el suministro utilizando el combustible en stock, de no tener stock suficiente tratará de refinar los combustibles hasta completar el suministro haciendo llamadas para cada uno de ellos, de ser necesario a la función `refinar()`.

La función retorna un diccionario con las cantidades reales suministradas. Según el stock y según lo que pueda refinar antes de agotarse las materias primas.

Ejemplo:

suministroReal = pedirSuministro({'gasolina95': 250, 'gasolina98': 300})

Valor suministroReal → {'gasolina95': 197, 'gasolina98': 300}

int refinar(combustible, litros)

Intentará refinar, según las fórmulas mostradas al principio, la cantidad pedida de litros.

Retorna la cantidad real refinada, puede darse el caso que nos quedemos sin materias primas.

infoMediaDias(dias)

Según el parámetro de días, que debe ser un entero (verificar su valor usando *Try – Except*, como los ejemplos vistos en clase), hace la media diaria de combustible que se puede suministrar para que el stock dure los días indicados.

Ejemplo:

infoMediaDias(10)

→

Suministro diario para 10 días:

Gasolina95 = 30

Gasolina98 = 27

Diesel = 45

Biodiesel = 29

bool recibirMateriaPrima(diccionario_materia_cantidad)

Análoga al caso de la gasolinera con las materias primas, se repone de materias primas la refinería hasta, como máximo, la capacidad de cada una de ellas.

Retorna *True* si todo va bien. *False*, si se produce un error o los tanques llegan a su máxima capacidad.

9.1.3. Componente 3: Integración componentes

Integrar componente1 y componente2 en un solo programa principal que importe los dos módulos por separado.

La configuración inicial de la gasolinera y de la refinería deberán poderse cargar leyendo un **fichero**.

Añadir una opción de salvar el estado actual de la gasolinera y de la refinería en un **fichero**.



Fichero con el estado actual. Fuente: imagen bajada de pixabay, CC0 Creative Commons.

Para poder acceder a todas las funciones, se debe implementar un menú. Para facilitar la usabilidad se accederá a las diferentes opciones introduciendo el número correspondiente. En el caso de que el usuario indique una opción inexistente, debe notificarse y volver a mostrar el menú.

Ejemplo de Menú Inicial:

1. Cargar Configuración.
2. Operar Gasolinera.
3. Operar Refinería.
4. Guardar Configuración.
5. Salir.

Una vez seleccionado el menú, se le mostrará el submenú correspondiente según el ejemplo.

Ejemplo Submenús:

1. Cargar Configuración.
 1. Cargar fichero configuración gasolinera.
 2. Cargar fichero configuración refinería.
 3. Salir menú principal.
2. Operar Gasolinera.
 1. Poner gasolina.
 2. Pedir suministros.

3. Salir menú principal.
3. Operar Refinería.
 1. Pedir suministros.
 2. Media días.
 3. Salir menú principal.
4. Guardar Configuración.
 1. Guardar fichero configuración gasolinera.
 2. Guardar fichero configuración refinería.
 3. Salir menú principal.
5. Salir.

Glosario

Apuntador

Contiene la dirección de memoria de un dato que está en la memoria. También se le conoce como 'puntero'.

Clase

Una clase es la descripción de todos los objetos de un mismo tipo. No podemos interactuar con una clase directamente sin antes instanciarla como objeto. Es una entidad abstracta.

CPU

La unidad central de procesamiento (CPU) del inglés *Central Processing Unit*, es el corazón de un ordenador u otros dispositivos programables. Es el componente de hardware que realiza las operaciones básicas aritméticas, lógicas y de entrada/salida en que son descompuestas las instrucciones de los programas.

Depurador

Herramienta de programación que nos permite ver la ejecución en detalle, pudiendo consultar los valores de las variables en vivo, paso a paso del código realizado para poder detectar y corregir los errores.

Excepción

Error poco frecuente que se produce por un error o interrupción no esperada durante la ejecución del código. El sistema crea un objeto especial que contiene información "detallada" sobre el error y es pasado/lanzado para que el mismo programador, en su código o, en caso contrario, el sistema operativo, lo trate para intentar recuperarse y seguir con la ejecución. No siempre es posible recuperarse y, a veces, inevitablemente conlleva que el programa actual aborte o, en casos más graves, hasta puede provocar que el sistema operativo necesite reiniciarse.

Garbage Collector

Rutina interna de algunos lenguajes de programación que se encarga de eliminar definitivamente las variables no usadas de memoria para liberar espacio. La eliminación se hace siguiendo un algoritmo de optimización para no eliminar innecesariamente variables que pueden ser necesitadas en breve.

IEEE

El Instituto de Ingeniería Eléctrica y Electrónica (IEEE), del inglés *Institute of Electrical and Electronics Engineers*, es la mayor asociación profesional mundial de ingenieros dedicada a la estandarización e innovación de tecnología para el desarrollo en beneficio de la humanidad.

Interfaz

Del inglés *interface*, en el contexto de programación es la parte visible del programa. En el caso que nos refiramos a código, es la parte que otros programadores pueden ver de él, las funciones y las variables que son visibles y pueden ser llamadas o consultadas respectivamente.

Lenguaje de programación de alto nivel

Los lenguajes de programación de alto nivel permiten **abstraerse** de cómo se representan internamente los datos, cómo se manipulan y cómo se implementa la lógica de los algoritmos, permitiendo al programador centrarse en la tarea de resolver el problema. Posteriormente, se realizará una transformación del algoritmo de alto nivel a un conjunto de instrucciones, lógica y datos que permita su ejecución en el sistema.

Módulo

Un módulo agrupa un conjunto de funciones que, normalmente, abordan un problema específico. Es el pilar de la programación modular que consiste en dividir un programa en módulos o subprogramas.

Objeto

Instanciación de una clase. Es cuando asignamos a la clase valor a sus atributos con los de un objeto real de ese tipo, el cual queremos representar en nuestro programa para interactuar con él.

Packet

Es la agrupación de módulos en un mismo directorio para formar un conjunto con funcionalidades normalmente afines.

Programación estructurada

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y facilitar el desarrollo de los programas. Divide el código en dos partes principales: el núcleo (donde está la lógica) y las funciones (las piezas necesarias). Recomienda utilizar exclusivamente estructuras de código de secuencia, selección e iteración. Recomienda encarecidamente no utilizar instrucciones de salto (aun cuando el lenguaje de programación utilizado las soporte) para mejorar la trazabilidad del código y disminuir los errores. Permite mejorar la claridad, calidad y tiempo de desarrollo del software.

Programación Orientada a Objetos

La programación Orientada a Objetos (OO) es un paradigma de programación basado en un modelo donde tenemos entidades llamadas objetos que son las que contienen tanto los datos como las operaciones que se pueden hacer sobre ellos. La clase de cada objeto determina de qué datos y funciones se dispone. Las clases son entidades abstractas y es necesario instanciarlas (dotarlas de valores específicos para un determinado objeto "real") para poder usarlas.

Python

Python es un lenguaje de programación **interpretado**, no se compila para generar un ejecutable, sino que el intérprete de Python lo va interpretando línea por línea bajo demanda. Tiene una sintaxis que ayuda a que el código sea más comprensible. Es **multiparadigma**: Orientación a Objetos, Programación Imperativa y Programación Funcional. Usa **tipado dinámico**, que permite que una misma variable vaya cambiando de tipo durante la ejecución según la necesidad. Puede ser ejecutado en múltiples plataformas de software o hardware sin adaptaciones en el código, es **multiplataforma**.

RAM

La memoria de acceso aleatorio (RAM), del inglés *Random Access Memory*, es donde se almacenan los datos y el código que se están usando actualmente por los programas para facilitárselos con más rapidez a la CPU de lo que se tardaría en traerlo cada vez del disco duro. Su tamaño es superior a la memoria interna de la CPU, conocida como memoria caché, y menor que los dispositivos principales de almacenamiento como pueden ser los discos duros, pero tiene una velocidad elevada sin llegar nunca a la de la caché interna, aunque superior a los dispositivos de almacenamiento.

Enlaces de interés

Tutorial de Python En Línea

Interesante tutorial en línea gratuito de cómo programar en Python.

<http://mundogeek.net/tutorial-python/>

Python Software Foundation

Web de la fundación que desarrolla Python. Multitud de recursos: Manuales, guía de referencias, Software, etc.

<https://www.python.org/>

Documentación: <https://docs.python.org/3/>

Tutorial: <https://docs.python.org/3/tutorial/index.html>

Software: <https://pypi.python.org/pypi/spyder/>

Anaconda

Distribución de Python muy completa.

<https://www.anaconda.com/download/>

Codecademy

Curso básico interactivo de Python.

<https://www.codecademy.com/es/tracks/python>

Bibliografía

Referencias bibliográficas

Boehm, B. W. (1988). *A spiral model of software development and enhancement*. Journal Computer, 21(5), 61-72.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). *Does code decay? Assessing the evidence from change management data*. IEEE Transactions on Software Engineering, 27(1), 1-12.

Gomis, P. (2017). "Manual de la asignatura 04GIIN Fundamentos de Programación". Grado de Ingeniería Informática Universidad Internacional de Valencia.

Lehman, M. M., Meir M. (1980). *Programs, Life Cycles, and Laws of Software Evolution*. Proc. IEEE 68 (9): 1060-1076.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). *Metrics and laws of software evolution-the nineties view*. In *Software metrics symposium, 1997. Proceedings, fourth international* (pp. 20-32). IEEE.

Pigoski, T. M. (1996). *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing.

Pressman, R. (2010). *Ingeniería del Software-Enfoque Práctico* (7ª Ed.). Mc Graw Hill.

Zuras, D., Cowlishaw, M., Aiken, A., Applegate, M., Bailey, D., Bass, S., ... & Canon, S. (2008). *IEEE standard for floating-point arithmetic*. IEEE Std 754-2008, 1-70.

Bibliografía recomendada

Almeida, M. A. R. (1993). *Metodología de la programación: a través de Pseudocódigo*. MacGraw-Hill.

Balcázar, J. L. (1993). *Programación metódica* (pp. I-XVI). McGraw-Hill.

Gomis, P. (2017). Manual de la asignatura 04GIIN Fundamentos de Programación. Grado de Ingeniería Informática Universidad Internacional de Valencia.

González, R. Tutorial de Python 'Python para todos'. Recuperado de <http://mundogeek.net/tutorial-python/>

Universidad Internacional de Valencia. Virtual Labs - Spyder. Recuperado de <https://virtuallabs.viu.es/>

Agradecimientos

Autor

D. Roger Clotet Martínez

Departamento de Recursos para el Aprendizaje

D.ª Carmina Gabarda López

D.ª Cristina Ruiz Jiménez

D.ª Sara Segovia Martínez

