Tema 1:

Introducción a Python

***Objetivos del tema**: este tema realiza un repaso rápido al lenguaje de programación Python, apoyándonos para ello en lo que hemos aprendido en la asignatura de Programación del primer curso.*

CEU

INDICE:

1 Python: el lenguaje 4

2 Programación modular en Python 6

2.1 Importando paquetes y módulos 7

2.2 Creación de nuevos módulos 7

2.3 Módulos comunes 8

3 Tipos de datos en Python 9

3.1 Tipos de datos 9

Int 9
Float 9
Bool 9
Strings 9
List 10
Tuple 10
Set 10
Dictionary 10
Operadores en listas y tuplas 11
Operadores en conjuntos 12

3.2 Creación y Utilización de Objetos 13

3.3 Accessors vs mutators 14

3.4 Built-in classes in Python 14

3.5 Expresiones y operadores en Python 14

Operadores lógicos 15
Operadores de igualdad 15
Operadores de comparación 15
Operadores aritméticos 15
Operadores sobre secuencias 16
Operadores sobre conjuntos (sets) 17
Operadores sobre diccionarios (dicts) 17
Asignación de operadores extendidos 18

3.6 Precedencia de operadores en Python 18

4 Tipos enumerados 19

5 Control de flujo Python 21

5.1 Sentencias Condicionales 21

if then else 21

5.2 Bucles 22

Bucle while 22
Bucle for 23

5.3 Break y continue 23

5.4 Return 24

6 Funciones 25

6.1 Paso de parámetros 25

Valores por defecto en el paso de parámetros 26
Parámetros keyword 27

6.2 Funciones Built-in 27

7 Lectura de datos de consola 29

7.1 Entrada y Salida por consola 29

La función print 29

La función input 29

7.2 Entrada y Salida de ficheros 30

Leyendo un fichero 30

Escribiendo en un fichero 30

8 Manejo de excepciones 32

8.1 Tipos de excepción más comunes 32

8.2 Lanzando (raising) excepciones 32

9 Iteradores 34

10 Documentación de Python 35

10.1 Comentarios vs Documentación 35

10.2 Comentarios 35

10.3 Documentación (Docstrings) 36

11 Testing 39

12 Ejercicios 43

1 Python: el lenguaje

La mayor parte de las personas que han empleado alguna vez un ordenador probablemente en alguna ocasión hayan oído hablar de algunos de estos lenguajes de programación: C, Pascal, Cobol, Visual Basic, Java, Fortran, Python, JavaScript... y a una persona ya más introducida en ese mundillo posiblemente haya oído hablar de muchos otros: Oak, Prolog, CUDA, Dbase, JavaScript, Delphi, Simula, Smalltalk, Modula, Oberon, Ada, BCPL, Common LISP, Scheme... En la actualidad se podrían recopilar del orden de varios cientos de lenguajes de programación distintos, sino miles.

Cabe hacerse una pregunta: ¿Para qué tanto lenguaje de programación? Toda esta multitud de nombres puede confundir a personas no iniciadas que hayan decidido aprender un lenguaje, quienes tras ver las posibles alternativas no saben cuál escoger, al menos entre los del primer grupo, que por ser más conocidos deben estar más extendidos.

El motivo de esta disparidad de lenguajes es que cada uno ha sido creado para una determinada función, está especialmente diseñado para: facilitar la programación de un determinado tipo de problemas; garantizar seguridad de las aplicaciones; obtener una mayor facilidad de programación; conseguir un mayor aprovechamiento de los recursos del ordenador, etc. Estos objetivos son muchos de ellos excluyentes: el adaptar un lenguaje a un tipo de problemas hará más complicado abordar mediante este lenguaje la programación de otros problemas distintos de aquellos para los que fue diseñado. El facilitar el aprendizaje al programador disminuye el rendimiento y aprovechamiento de los recursos del ordenador por parte de las aplicaciones programadas en este lenguaje; mientras que primar el rendimiento y aprovechamiento de los recursos del ordenador tiende a dificultar labor del programador.

La pregunta que viene a continuación es evidente: ¿Para qué fue diseñado Python? A continuación, haremos una pequeña relación de las características del lenguaje, que nos ayudarán a ver para qué tipo de problemas está pensado Python:

- **Simple**: es un lenguaje sencillo de aprender. La herencia múltiple, la aritmética de punteros, o la gestión de memoria dinámica (en Python no hace falta pensar en términos de reservar y liberar memoria) son ejemplos de "tareas complicadas" de C/C++ que en Python se han eliminado o simplificado.
- **Orientado a objetos**: en Python todo es un objeto.
- **Distribuido:** Python está muy orientado al trabajo en red, soportando protocolos como TCP/IP, UDP, HTTP y FTP. El uso de estos protocolos es bastante sencillo comparándolo con otros lenguajes que los soportan: Socket, Request, Asyncio o Diesel son ejemplos de librerías usadas para el desarrollo de trabajo en red.

- **Interpretado:** cuando se ejecuta Python, no se ejecuta directamente, sino que el interprete Python lo convierte a código máquina que es el que se ejecuta.
- **Multiparadigma:** aunque se diseñó especialmente para la programación orientada a objetos, existen otros paradigmas como la programación imperativa o la programación funcional.
- **Multiplataforma**: existen intérpretes para muchos dispositivos y sistemas operativos (Unix, Linux, MacOs, Windows, etc.).
- **Tipado dinámico**: en Python el objetivo es que el lenguaje ayude a la creación de software, no tener que lidiar con peculiaridades propias del lenguaje, por lo que prioriza la elegancia en la escritura. Sin embargo, Python es **fuertemente tipado**, por ejemplo, no podrás sumar números y texto (una variable del tipo int con una de tipos char) porque ocurriría un error.
- **Rendimiento medio**: Python no es un lenguaje tan rápido o eficiente como los lenguajes compilados (C o Java, que permite el uso de compiladores just-in-time). Tampoco es un lenguaje especialmente diseñado para el ahorro en el uso de memoria.
- **Multithread:** soporta los threads, mediante uso de librerías específicas (como map o pool). Esto le permite además que cada Thread de una aplicación Python pueda correr en una CPU (o core) distinta, si la aplicación se ejecuta en una máquina que posee varias CPU.

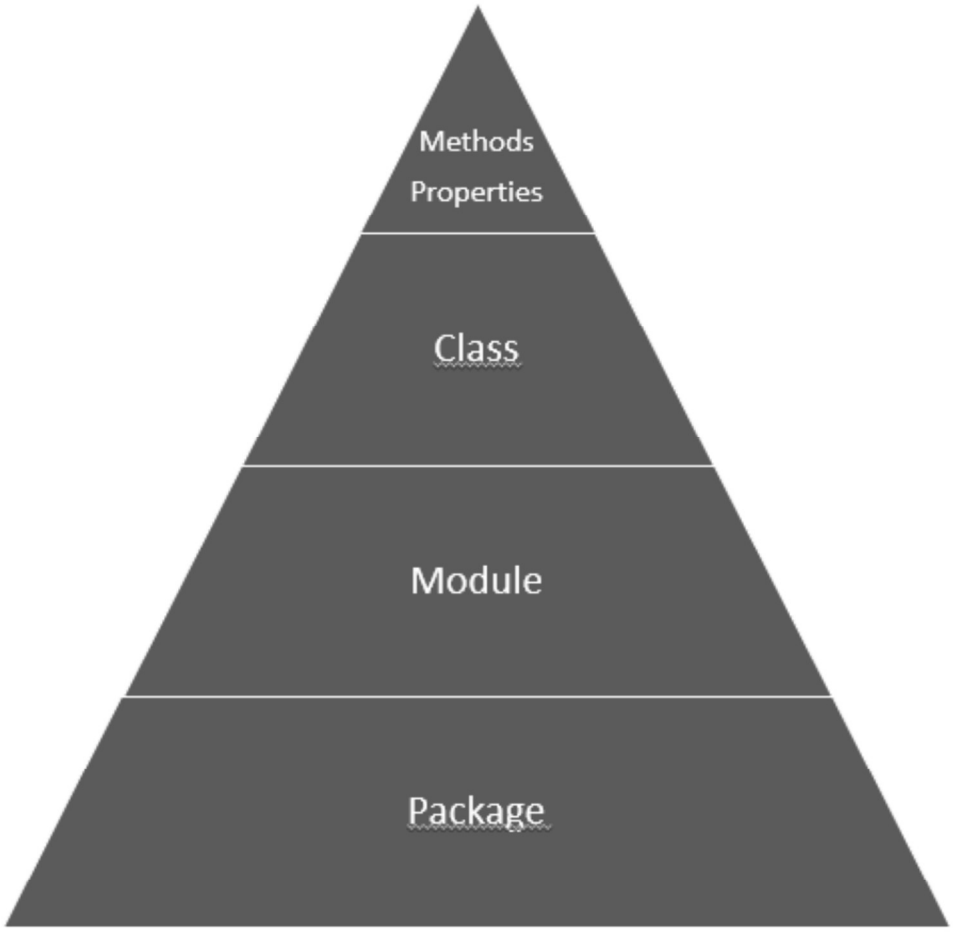
2 Programación modular en Python

La programación modular se refiere al proceso de **dividir un problema muy grande en diferentes tareas módulos más pequeños y manejables.** Los módulos o subtareas independientes pueden luego ser unidos para resolver el problema inicial y más grande. Una metáfora utilizada habitualmente es la construcción de edificios. Para ello, se comienza con los cimientos, tras ello se van realizando módulos independientes y al poner todos juntos se construye la casa o edificio completo.

Existen múltiples ventajas de la programación modular cuando se quieren crear aplicaciones grandes:

- **Simplicidad:** permite enfocarse en solucionar pequeñas porciones de un problema grande.
- **Mantenibilidad:** los módulos se diseñan para tener una funcionalidad independiente del resto de la aplicación. Si se consigue que los módulos sean independientes, esto hace que los cambios en un módulo no afecten al resto de módulos que forman una aplicación, y permite que se pueda modificar un módulo sin tener conocimiento de los demás.
- **Reusabilidad:** la funcionalidad de un módulo puede ser reutilizada en otras partes de la aplicación. Esto elimina la necesidad de crear código duplicado.
- **Ámbito:** los módulos se definen típicamente en un **namespace** diferente, lo que ayuda a evitar colisiones entre identificadores en diferentes módulos de la aplicación.

La siguiente figura representa la jerarquía de paquetes/módulos/clases y métodos en Python. Un paquete puede contener 0 o más módulos, un módulo 0 o más clases, una clase 0 o más métodos y propiedades.



2.1 Importando paquetes y módulos

Para importar un paquete y todos sus módulos se puede realizar de cualquiera de las siguientes formas:

```
import package_name
```

```
from package_name import *
```

Para importar un módulo de un paquete se debe especificar el nombre del paquete:

```
import package_name.module_name
```

```
from package_name import module_name
```

2.2 Creación de nuevos módulos

Para crear un nuevo módulo, simplemente debemos crear un nuevo fichero .py, y este módulo puede ser importado desde cualquier fichero en el mismo directorio. Por ejemplo, si hemos creado nuestro

módulo equation.py, que contiene una función que se llama second_grade_equation se puede importar utilizando la sentencia

```
from equation import second_grade_equation
```

Cuando un módulo debe ser ejecutado, este debe contener la función especial main(), y así podrá ser invocado como script. Para hacer nuestros programas compatibles con versiones anteriores de Python 3, incluiremos siempre en nuestros programas de Python ejecutables el siguiente comando:

```
if __name__ == "__main__":
  main()
```

2.3 Módulos comunes

Existen una serie de módulos comúnmente utilizados, que se describen a continuación:

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

3 Tipos de datos en Python

En este tema trataremos las estructuras básicas de Python de datos, sus tipos y las variables, operadores. Aquellos que estén familiarizados con C, o C++, no encontrarán prácticamente nada nuevo en este tema.

3.1 Tipos de datos

En Python toda variable declarada ha de tener su identificador, pero su tipo de datos es dinámico, y va a tomar el tipo según el valor asignado, por ello, no se pueden tener variables sin inicializar. A continuación, pasamos a describir los tipos de datos primitivos soportados en Python. Cabe destacar que el número de bites asignados a los tipos de datos en Python dependerá de la arquitectura en la que se esté ejecutando y de la longitud de los mismos. Por tanto, los tipos de datos no tienen una longitud fija y contiene mucha más información que el propio valor del dato, ya que todos ello son objetos, como veremos más adelante.

Int

Almacenan como su propio nombre indica números enteros, sin parte decimal.

Float

Almacenan número reales, es decir números con parte fraccionaria.

Bool

Se trata de un tipo de dato que solo puede tomar dos valores: "**True**" y "**False**". Es un tipo de dato bastante útil a la hora de realizar chequeos sobre condiciones. En C, hasta C99, no había un dato equivalente y para suplir su ausencia muchas veces se emplean enteros con valor 1 si "true" y 0 si "false". Otros lenguajes como Pascal sí tienen este tipo de dato.

Strings

Secuencias de caracteres.

```
myString = "I am a string"
```

List

Colecciones **ordenadas y mutables** de datos de **diferentes** tipos. Se declara de la siguiente forma:

```
numeros = [1,2,3]
```

Métodos built-in: min, max, len, append, insert, remove, pop, reverse, sort, list, tuple.

Operadores: Concatenación, repeticion, slice, range slice, in, not in.

Tuple

Colecciones **ordenadas e inmutables** de datos de **diferentes** tipos.

```
numeros = (1,2,3)
```

Métodos built-in: min, max, len.

Operadores: Concatenación, repeticion, slice, range slice, in, not in.

Set

Colecciones **desordenadas y mutables** de datos de **diferentes** tipos **inmutables**. Esto quiere decir que los objetos mutables (listas o diccionarios) no pueden formar parte de un conjunto. Representa un conjunto matemático

```
numeros = {1,2,3}
```

Métodos built-in: add, update, clear, copy, discard, remove

Operadores: Union(|), intersection(&), difference(-), Symmetric_difference(^),

Dictionary

Colecciones **desordenadas** de **parejas clave-valor**.

```
dict = { key1:value1, key2:value2,...keyN:valueN }
```

La clave debe ser un objeto **inmutable**. Esto quiere decir que una lista u otro diccionario no pueden ser la clave de un elemento del diccionario. La misma clave no puede aparecer dos veces en un mismo diccionario. Si se intenta insertar una pareja clave-valor en un diccionario que ya contiene dicha clave, se actualizará el valor de la clave y se perderá el valor antiguo.

Métodos built-in: min, max, len, pop, clear, items, keys, values, update.

Operadores: ninguno.

Operadores en listas y tuplas

Operator	Description	Example

+ Concatenation	Returns a list containing all the elements of the first and the second list.	>>> L1=[1,2,3]
		>>> L2=[4,5,6]
		>>> L1+L2
		[1, 2, 3, 4, 5, 6]

* Repetition	Concatenates multiple copies of the same list.	>>> L1*4 [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

[] slice	Returns the item at the given index. A negative index counts the position from the right side.	>>> L1=[1, 2, 3, 4, 5, 6]
		>>> L1[3]
		4
		>>> L1[-2]
		5

[:] - Range slice	Fetches items in the range specified by the two index operands separated by : symbol.
	If the first operand is omitted, the range starts from the zero index. If the second operand is omitted, the range goes up to the end of the list.
		>>> L1=[1, 2, 3, 4, 5, 6]
		>>> L1[1:4]
		[2, 3, 4]
		>>> L1[3:]
		[4, 5, 6]
		>>> L1[:3]
		[1, 2, 3]

in	Returns true if an item exists in the given list.	>>> L1=[1, 2, 3, 4, 5, 6]
		>>> 4 in L1
		True
		>>> 10 in L1
		False

not in	Returns true if an item does not exist in the given list.	>>> L1=[1, 2, 3, 4, 5, 6]
		>>> 5 not in L1
		False
		>>> 10 not in L1
		True

Operadores en conjuntos

Operation	Operator/Method	Example

The union of two sets is a set of all elements from both the collections.	|	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1|s2
		{1, 2, 3, 4, 5, 6, 7, 8}

	union()	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1.union(s2)
		{1, 2, 3, 4, 5, 6, 7, 8}
		>>> s2.union(s1)
		{1, 2, 3, 4, 5, 6, 7, 8}

The intersection of two sets is a set containing elements common to both collections.	&	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1&s2
		{4, 5}
		>>> s2&s1
		{4, 5}

	intersection()	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1.intersection(s2)
		{4, 5}
		>>> s2.intersection(s1)
		{4, 5}

The difference of two sets results in a set containing elements only in the first set, but not in the second set.	-	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1-s2
		{1, 2, 3}
		>>> s2-s1
		{8, 6, 7}

	difference()	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1.difference(s2)
		{1, 2, 3}
		>>> s2.difference(s1)
		{8, 6, 7}

Symmetric Difference: the result of symmetric difference is a set consisting of elements in both sets, excluding the common elements.	^	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1^s2
		{1, 2, 3, 6, 7, 8}
		>>> s2^s1
		{1, 2, 3, 6, 7, 8}

	Symmetric_difference()	>>> s1={1,2,3,4,5}
		>>> s2={4,5,6,7,8}
		>>> s1.symmetric_difference(s2)
		{1, 2, 3, 6, 7, 8}
		>>> s2.symmetric_difference(s1)
		{1, 2, 3, 6, 7, 8}

3.2 Creación y Utilización de Objetos

El proceso de creación de una nueva instancia de una clase (**recordemos aquí que todas las variables son objetos en Python**), se conoce como instanciación. La sintaxis de inicialización de Python requiere de una inicialización.

```
students = 20
```

o

```
students = int('20',10)
```

La sintaxis de la clase int es la siguiente:

```
int(string, base)
```

Donde la base del número se puede cambiar a binario, 3, octal, decimal, hexadecimal, etc.) Por defecto, la base es decimal.

Para invocar/llamar a los métodos de un objeto, se utiliza:

```
welcome = "Welcome to my weirdo world."
```

o

```
welcome = "Welcome to my weirdo world."
lower_welcome = welcome.lower()
```

En Python, al igual que en todo lenguaje de programación hay una serie de palabras reservadas que no pueden ser empleadas como nombres de variables (if, int, char, else, goto....); alguna de estas se emplean en la sintaxis del lenguaje, otras, como goto no se emplean en la actualidad pero se han reservado por motivos de compatibilidad por si se emplean en el futuro. Esta es la lista de palabras reservadas:

Reserved Words								
False	as	continue	else	from	in	not	return	yield
None	assert	def	except	global	is	or	try	
True	break	del	finally	if	lambda	pass	while	
and	class	elif	for	import	nonlocal	raise	with	

Los caracteres aceptados en el nombre de una variable son los comprendidos entre “A-Z”, “a-z”, _, $ y cualquier carácter que sea una letra en algún idioma, no necesariamente sólo del idioma inglés, como sucede en C. No obstante, es recomendable limitarse a emplear caracteres del alfabeto inglés para curarse en salud.

3.3 Accessors vs mutators

Los métodos que devuelven información acerca del estado de un objeto se conocen como **accessors**, ya que el objeto no cambia sin importar las veces que se invoquen, mientras que los métodos que realizan acciones que cambian el estado de un objeto se conocen como **mutators**.

Un ejemplo de un método accessor es el método count, que devuelve el número de apariciones de una subcadena dentro de una cadena, sin cambiar el estado del objeto.

Un ejemplo de un método mutator sería un método set que modificase cualquier propiedad del objeto, en el caso de las string no hay métodos mutators ya que todas devuelven un nuevo objeto.

3.4 Built-in classes in Python

Este es un resumen de las clases preconstruidas en Python. La columna immutable hace referencia a si las clases son mutables o, por el contrario, una vez se crea un objeto de una clase este no puede modificar su estado. Las únicas clases mutables en Python son las listas, los conjuntos y los diccionarios.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Para definir tuplas se utilizan paréntesis

```
numeros = (1,2,3)
numeros = 1,2,3
```

Para definir listas se utilizan corchetes

```
numeros = [1,2,3]
```

3.5 Expresiones y operadores en Python

En las secciones anteriores se demostró cómo se pueden instanciar objetos, acceder a sus métodos y definirlos, y cómo los constructores pueden ser usados para la construcción de las clases predefinidas en Python. Estas variables pueden ser combinadas utilizando expresiones sintácticas conocidas como operadores.

Operadores lógicos

- **not:** negación unaria.
- **and:** condicional.
- **Or:** condicional.

Los operadores lógicos and y or no evalúan los operadores subsecuentes una vez que el resultado puede ser determinado basado en los anteriores. Por ejemplo:

```
fal1 = False
fal2 = False
tru1 = True
tru2 = True
tru1 or fal1
```

Esta expression solo evalúa la primera condición y, cómo se cumple, no evalúa la segunda.

Operadores de igualdad

- **is:** misma identidad. Los dos identificadores apuntan al **mismo** objeto.
- **is not:** diferente identidad. Los dos identificadores apuntan a **diferentes** objetos.
- **==** equivalente. Los dos identificadores apuntan a **objetos con el mismo valor**
- **¡=** no equivalente. Los dos identificadores apuntan a **objetos con diferente valor**.

Operadores de comparación

- **<** menor que
- **<=** menor o igual que
- **>** mayor que
- **>=** mayor o igual que.

Se producen excepciones cuando se comparan tipos incompatibes (e.g. un string y un double).

Operadores aritméticos

- **+** suma
- - resta
- ***** multiplicación

- **/** división
- **//** división entera
- **%** módulo.

La multiplicación, división, división entera y módulo tienen preferencia sobre la suma y la resta, tal y como ocurre en las matemáticas. Si se desea una preferencia distinta se debe especificar mediante el uso de paréntesis.

Operadores sobre secuencias

Python tiene tres tipos de secuencias built-in: las strings, las tuplas y las listas, que soportan las siguientes operaciones (siendo s y t dos secuencias built-in del mismo tipo):

- **s[j]** elemento en el índice j
- **s[start:stop]** devuelve los elementos desde start hasta stop incluyendo el start y no el stop.
- **s[start:stop:step]** devuelve los elementos start, start + step, start + 2*step, etc.. hasta stop, sin incluir stop.
- **s + t** concatenación de secuencias s y t
- **k * s** siendo k un entero: concatenación de s un número k de veces (s + s + s … hasta k veces)
- **val in s** devuelve True si el valor val está en s. False en caso contrario
- **val not in s** devuelve True si el valor val no está en s. False en caso contrario

Python indexa las secuencias desde 0 hasta n-1, ambos incluidos, siendo n la longitud de la secuencia.

Las secuencias soportan los operandos de igualdad y comparación:

- s == t
- s != t
- s < t
- s <= t
- s > t
- s >= t

Operadores sobre conjuntos (sets)

Python tiene dos tipos de conjuntos built-in: set y frozenset, que soportan las siguientes operaciones (siendo s1 y s2 dos conjuntos built-in del mismo tipo). En Python, los conjuntos no tienen un orden definido

Nota: un subconjunto o superconjunto es apropiado si los conjuntos no son exactamente iguales.

- **key in s** comprobación de clave key en el conjunto o diccionario s.
- **key not in s** comprobación de no-clave key en el conjunto o diccionario s.
- **s1 == s2** s1 equivalente a s2
- **s1 != s2** s1 no equivalente a s2
- **s1 <= s2** s1 es subset de s2
- **s1 < s2** s1 es subset apropiado de s2
- **s1 >= s2** s1 es superset de s2
- **s1 > s2** s1 es superset apropiado de s2
- **s1 | s2** unión de s1 y s2
- **s1 & s2** intersección de s1 y s2
- **s1 – s2** conjunto de elementos de s1 que no están en s2
- **s1 ^ s2** conjunto de elementos que están en s1 o s2, pero no en ambos.

Operadores sobre diccionarios (dicts)

Los diccionarios, al igual que los conjuntos, no mantienen un orden definido, y soportan las siguientes operaciones (siendo d d1 y d2 dicts).

- **d[key]** valor asociado a la clave key.
- **d[key] = value** establecer o restableces el valor asociado a la clave key con el valor value.
- **del d[key]** eliminar la clave key y su valor asociado en el diccinario.
- **key in d** comprobación de la clave key en el diccionario d.
- **key not in d** comprobación de no clave key en el diccionario d.
- **d1 == d2** d1 es equivalente a d2 (mismas claves, mismos valores)
- **d1 != d2** d1 no es equivalente a d2 (mismas claves, mismos valores)

Asignación de operadores extendidos

En la asignación de listas, es importante distinguir entre las operaciones:

```
list1 = [1,2]
list2 = list1   #alias para list1 (dos identifiers apuntando
al mismo objeto).
list2 += [3,4] #extiende la lista original con dos elementos
#(se utiliza un método mutator)
list2 = list2 + [5,6] #reasigna list2 a un nuevo objeto list
```

3.6 Precedencia de operadores en Python

En la siguiente table se muestra la lista de operadores en Python y su orden de preferencia de mayor a menor.

Operator Precedence		
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ~expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	comparisons containment	**is**, **is not**, ==, !=, <, <=, >, >= **in**, **not in**
12	logical-not	**not** expr
13	logical-and	**and**
14	logical-or	**or**
15	conditional	val1 **if** cond **else** val2
16	assignments	=, +=, −=, *=, etc.

4 Tipos enumerados

Los tipos de datos enumerados son un tipo de dato definido por el programador (no como ocurre con los tipos de datos primitivos). En su definición el programador debe indicar un conjunto de valores finitos sobre los cuales las variables de tipo enumeración deberán tomar valores. La principal funcionalidad de los tipos de datos enumerados es incrementar la legibilidad del programa. La mejor forma de comprender lo qué son es viéndolo; para definir un tipo de dato enumerado se emplea la sintaxis:

```
from enum import Enum
class Color(Enum):
  RED = 1
  GREEN = 2
  BLUE = 3
...
# Printing enumerations
print(Color.Red)
# Printing detailed information of enumerations
print(repr(Color.Red))
```

Los posibles valores de "modificadores" serán vistos más adelante. El caso más habitual es que modificadores tome el valor "public", que es su valor por defecto. El nombre puede ser cualquier nombre válido dentro de Python. Entre las llaves se ponen los posibles valores que podrán tomar las variables de tipo enumeración, valores que habitualmente se escriben en letras mayúsculas. Un ejemplo de enumeración podría ser:

```
Animal = Enum('Animal', 'ANT BEE CAT DOG')
```

Las definiciones de los tipos enumerados deben realizarse fuera del método main y, en general, fuera de cualquier método; es decir, deben realizarse directamente dentro del cuerpo de la clase. Más adelante se explicará detalladamente qué es una clase y qué es un método.

Para definir una variable de la anterior enumeración se emplearía la siguiente sintaxis:

```
ant = Animal.ANT
cat = Animal.CAT
```

Veamos un ejemplo de programa que emplea enumeraciones:

```
from enum import Enum
class Semana(Enum):
  LUNES = 1
  MARTES = 2
  MIERCOLES = 3
  JUEVES = 4
  VIERNES = 5
  SABADO = 6
  DOMINGO = 7

def main():
   #creating 2 new objects  from the Semana Enum
   lunes=Semana.LUNES
   martes=Semana.MARTES
   # printing an enum value
   print(lunes,martes)
   # calling a mutator method


if __name__ == "__main__":
    main()
```

5 Control de flujo Python

El modo de ejecución de un programa en Python en ausencia de elementos de control de flujo es secuencial, es decir una instrucción se ejecuta detrás de otra y sólo se ejecuta una vez. Esto nos permite hacer programas muy limitados; para evitarlo se introducen estructuras de control de flujo. Las estructuras de control de flujo de Python son las típicas de cualquier lenguaje de programación, por lo que supondremos que todos estáis familiarizados con ellas y se explicaran con poco detenimiento.

5.1 Sentencias Condicionales

Ejecutan un código u otro en función de que se cumpla o no una determinada condición. Pasemos a ver sus principales tipos.

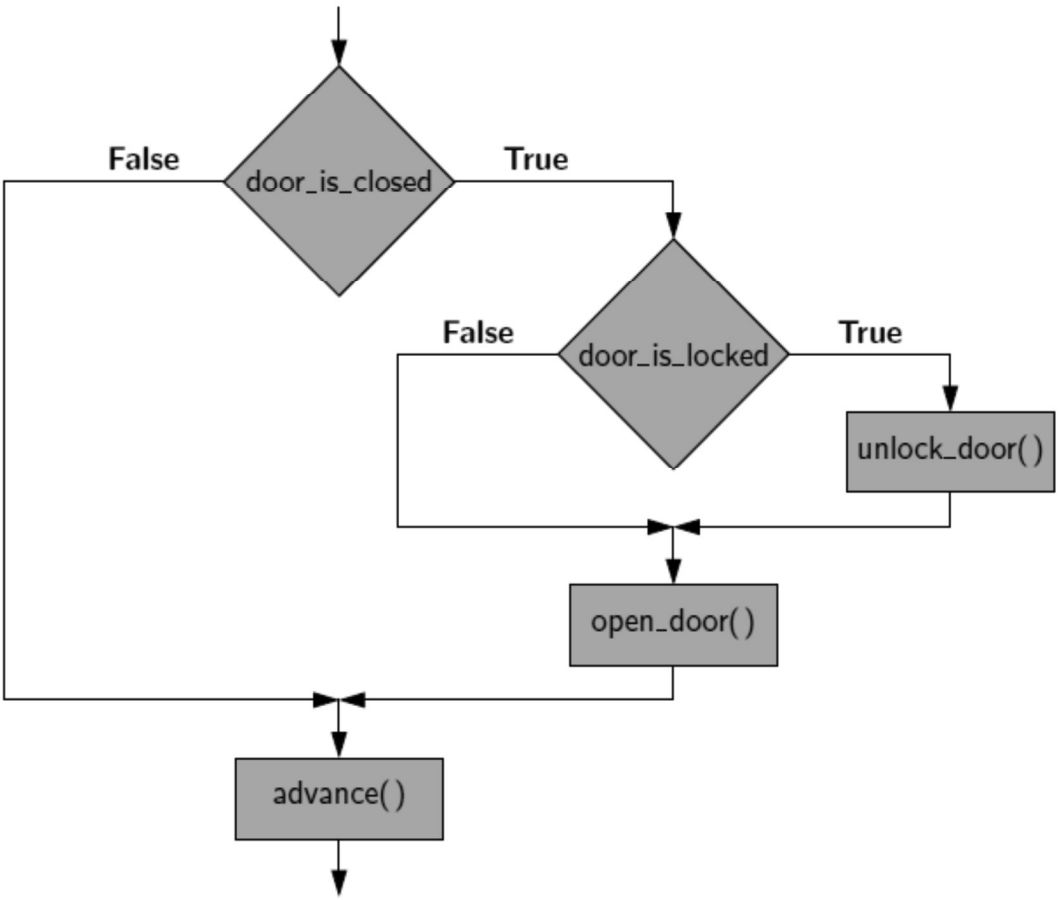
if then else

Su modo más simple de empleo es:

```
if first_condition:
  first_body
elif second_conditiong:
  second_body
…
else:
  last_body
```

Cada condición es un valor tipo boolean, y cada cuerpo contiene una o más sentencias para ser ejecutadas condicionalmente. El grupo de sentencias se ejecuta solo si la condición toma un valor true. En caso contrario se sigue ejecutando ignorando el grupo de sentencias.

```
if door_is_closed:
  if door_is_locked:
    unclock_door()
  open_door()
advance()
```



5.2 Bucles

Son instrucciones que nos permiten repetir un bloque de código mientras se cumpla una determinada condición. Pasemos a ver sus tipos.

Bucle while

Cuando en la ejecución de un código se llega a un bucle while se comprueba si se verifica su condición, si se verifica se continúa ejecutando el código del bucle hasta que esta deje de verificarse. Su sintaxis es:

```
while condition:
  body
```

Ilustramos su funcionamiento con un ejemplo:

```
data = "String that will be read until a X character
appears"j = 0
while j < len(data) and data[j] != 'X' :
  j += 1
```

Bucle for

Su formato es el siguiente:

```
for element in iterable:
  body
```

Veamos un ejemplo:

```
suma = 0
for number in list_numbers:
  suma = suma + number
```

El bucle for en Python es muy simple. Sin embargo, este bucle no permite conocer el orden de los elementos en una secuencia. En algunas aplicaciones, conocer el índice de un elemento es necesario, como por ejemplo buscando el máximo elemento de una lista.

En lugar de iterar directamente sobre los elementos, se puede iterar sobre los índices del elemento y acceder a ellos. Por ejemplo:

```
big_index = 0
for j in len(list_numbers)):
  if(list_numbers[j] > list_numbers[big_index]:
    big_index=j
```

5.3 Break y continue

No se tratan de un bucle, pero sí de sentencias íntimamente relacionadas con estos. El encontrarse una sentencia break en el cuerpo de cualquier bucle detiene la ejecución del cuerpo del bucle y sale de este, continuándose ejecutando el código que hay tras el bucle.

Esta sentencia también se puede usar para forzar la salida del bloque de ejecución de una instrucción condicional.

```
found = False
for item in data:
  if item == target:
    found = True
    break
```

Continue también detiene la ejecución del cuerpo del bucle, pero en esta ocasión no se sale del bucle, sino que se pasa a la siguiente iteración de este. Se recomienda el uso muy esporádico de estas instrucciones, pero aún hay situaciones donde estos comandos pueden ser efectivos y eficientes para evitar introducir condiciones lógicas complejas.

Observaremos el funcionamiento de ambos en un mismo ejemplo que cuenta las apariciones de target en la secuencia data.

```
def count(data, target):
  n = 0
  for item in data:
    if item == target: # found a match
      n += 1
  return n
```

5.4 Return

Cuando se llama a un procedimiento (que en OOP se denominó método) al encontrarse con una sentencia return se pasa el valor especificado al código que llamó a dicho método y se devuelve el control al código invocador. Su misión tiene que ver con el control de flujo: se deja de ejecutar código secuencialmente y se pasa al código que invocó al método.

Esta sentencia también está profundamente relacionada con los métodos, ya que es la sentencia que le permite devolver al método un valor. Podíamos haber esperado a hablar de métodos/funciones para introducir esta sentencia, pero hemos decidido introducirla aquí por tener una función relacionada, entre otras cosas, con control de flujo.

6 Funciones

En esta sección se explicará la creación y el uso de funciones en Python. Se van a distinguir las funciones de los métodos en que los métodos van a ser funciones que forman parte de una clase, mientras que las funciones no van a estar asociadas a ninguna clase. Se van a distinguir fácilmente porque los métodos sólo pueden ser llamados desde un objeto, mientras que las funciones no van a necesitar de ningún objeto para ser ejecutadas.

6.1 Paso de parámetros

Para poder programar correctamente, se debe entender de forma clara el mecanismo de paso de información desde y hacia una función. Para pasar información a una función se utilizan parámetros, y para devolver información desde una función se utiliza la sentencia return, al igual que se pueden modificar los objetos mutables que se pasan como parámetro. La signatura de una función se refiere a los parámetros formales que una función espera cuando es invocada (deben ser especificados en la documentación), mientras que los parámetros reales son los que realmente se envían en la llamada a la función. Se puede y se debe hacer comprobación de tipos de parámetros cuando es necesario para el control de errores, que se debe manejar mediante el uso de excepciones.

Para verlo con un ejemplo, se puede utilizar la función:

```
def count(data, target):
  n = 0
  for item in data:
    if item == target: # found a match
      n += 1
  return n
```

Esta función espera una secuencia iterable y un objeto del tipo de objetos de la secuencia iterable para contar el número de apariciones del objeto en la secuencia. Para invocar la función se utiliza:

```
prizes = count(grades, 'A')
```

Los parámetros reales son grades y 'A', mientras que los parámetros formales son data y target, y la función hace una asignación de los parámetros reales con los formales:

data = grades

target = 'A'

En caso de que el parámetro enviado no sea una secuencia, la función va a devolver una excepción al intentar iterar un objeto no iterable. Por ello, es muy importante documentar qué tipos de datos esperan los parámetros formales de una función. Lo mismo ocurre con el tipo de datos devuelto por una función.

Cuando el tipo de datos de un parámetro es un objeto mutable, se puede interaccionar con estos objetos y los cambios serán guardados una vez el control se devuelve al lugar desde donde se invocó la función. Por ejemplo, si se quiere escalar una lista de números por un factor, se puede hacer mediante la función:

```
def scale(data, factor):
  for j in range(len(data)):
    data[j] *= factor
```

Y la lista data (parámetro real) contendrá los nuevos valores escalados por el factor especificado por el usuario (parámetro real).

Valores por defecto en el paso de parámetros

Python soporta la llamada a una función con diferente número de parámetros. A esta propiedad se le llama polimorfismo. Se pueden especificar valores por defecto en la signatura de una función, y al llamar a esa función con un número de parámetros menor a la signatura, la función toma los valores especificados por defecto. Veamos esto con un ejemplo.

```
def scale(data, factor=2):
  for j in range(len(data)):
    data[j] *= factor
# Call to a function
my_data=[1,2,3,4]
# Method of list that returns a copy of the list where it is
called
my_data2= my_data.copy()
scale(my_data)
print(my_data)
scale(my_data2,4)
print(my_data2)
```

Como se aprecia, el usuario puede invocar la función con al menos un parámetro, y los parámetros con un valor por defecto son opcionales. **Por ejemplo, se puede definir un sistema de evaluación por defecto, que computa los grados desde A+ hasta F, y calcula la media. En caso de que haya algún grado en la lista que no aparezca en el sistema de evaluación, este se ignorará, sin devolver ningún error.**

```
def compute_gpa(grades, points={'A+':4.0,'A':4.0,'A-
':3.67,'B+':3.33,'B':3.0,'B-':2.67,'C+':2.33,'C':2.0,'C-
':1.67,'D+':1.33,'D':1.0,'F':0.0}):
  num courses = 0
  total points = 0
  for g in grades:
    if g in points: # a recognizable grade
      num courses += 1
      total points += points[g]
  return total points / num courses
```

Parámetros keyword

Python también soporta la llamada mediante el uso de keywords. Por ejemplo, la función:

```
def whatever(a=1,b=2,c=3):
  body
```

Puede ser llamada utilizando las keywords y con todas las combinaciones posibles: whatever(b=4), ejecutará la función con los valores por defecto a=1 y c=3 y el parámetro real invocado de b=4, y así con todas las combinaciones.

Los parámetros que no contienen un valor por defecto son obligatorios en la llamada a la función y deben ser declarados antes que los parámetros opcionales.

6.2 Funciones Built-in

Python proporciona una serie de funciones comunes que están disponibles automáticamente en la instalación de Python. Estas son algunas de las más importantes o que van a ser utilizadas en este curso:

- **Entrada/Salida**: print, input and open, entre otras, para imprimir en consola, leer de teclado o abrir un fichero, respectivamente.
- **Character Encoding**: ord and chr, para relacionar los caracteres y su código ASCII.
- **Matemáticas:** abs, divmod, pow, round or sum, entre otras. El nombre es descriptivo de lo que hacen estas funciones.
- **Ordenación:** max or min son dos funciones que calculan el máximo y el mínimo en una colección. También hay funciones para ordenar (sort) de acuerdo a un criterio.
- **Colecciones/Iteraciones:** len, iter y next.

Y en la siguiente tabla se describen las funciones built-in más comunes:

Common Built-In Functions	
Calling Syntax	**Description**
abs(x)	Return the absolute value of a number.
all(iterable)	Return True if bool(e) is True for each element e.
any(iterable)	Return True if bool(e) is True for at least one element e.
chr(integer)	Return a one-character string with the given Unicode code point.
divmod(x, y)	Return (x // y, x % y) as tuple, if x and y are integers.
hash(obj)	Return an integer hash value for the object (see Chapter 10).
id(obj)	Return the unique integer serving as an "identity" for the object.
input(prompt)	Return a string from standard input; the prompt is optional.
isinstance(obj, cls)	Determine if obj is an instance of the class (or a subclass).
iter(iterable)	Return a new iterator object for the parameter (see Section 1.8).
len(iterable)	Return the number of elements in the given iteration.
map(f, iter1, iter2, ...)	Return an iterator yielding the result of function calls f(e1, e2, ...) for respective elements e1 ∈ iter1, e2 ∈ iter2, ...
max(iterable)	Return the largest element of the given iteration.
max(a, b, c, ...)	Return the largest of the arguments.
min(iterable)	Return the smallest element of the given iteration.
min(a, b, c, ...)	Return the smallest of the arguments.
next(iterator)	Return the next element reported by the iterator (see Section 1.8).
open(filename, mode)	Open a file with the given name and access mode.
ord(char)	Return the Unicode code point of the given character.
pow(x, y)	Return the value x^y (as an integer if x and y are integers); equivalent to x ** y.
pow(x, y, z)	Return the value $(x^y \bmod z)$ as an integer.
print(obj1, obj2, ...)	Print the arguments, with separating spaces and trailing newline.
range(stop)	Construct an iteration of values $0, 1, \ldots,$ stop $- 1$.
range(start, stop)	Construct an iteration of values start, start $+ 1, \ldots,$ stop $- 1$.
range(start, stop, step)	Construct an iteration of values start, start + step, start + 2*step, ...
reversed(sequence)	Return an iteration of the sequence in reverse.
round(x)	Return the nearest int value (a tie is broken toward the even value).
round(x, k)	Return the value rounded to the nearest 10^{-k} (return-type matches x).
sorted(iterable)	Return a list containing elements of the iterable in sorted order.
sum(iterable)	Return the sum of the elements in the iterable (must be numeric).
type(obj)	Return the class to which the instance obj belongs.

7 Lectura de datos de consola

En esta sección, se especifican las operaciones básicas de entrada y salida por consola y de icheros.

7.1 Entrada y Salida por consola

La función print

La función print se utiliza como la salida standard por consola. La función print imprime una secuencia de argumentos enviados durante la invocación de la función. Los argumentos no tienen por qué ser Strings. Por defecto, el método print utilizado el espacio para separar los diferentes argumentos, aunque se puede modificar con la keyword sep:

```
print(a, b, c, sep=':', end='END')
```

En este ejemplo los parámetros a b y c serán separados por dos puntos en lugar de espacios, y al final del texto se imprimirá END en lugar de un salto de línea. Por defecto el método print imprime en consola, pero puede ser redireccionado a ficheros u otros streams especificando el keyword parameter 'file=file_name'.

La función input

A continuación, podremos leer líneas de texto empleando el método input (). Por defecto, el método input lee de teclado y recoge la entrada como una cadena de caracteres seguido de un retorno de carro. La entrada es toda la secuencia de caracteres introducida antes de este retorno de carro.

```
stringLeido = input('Introduzca dos numeros, separados por
espacios')
my_floats = stringLeido.split( )
float1 = float(my_float[0])
float2 = float(my_float[1])
```

A continuación, se muestra un ejemplo de un programa que lee la edad del usuario por consola y calcula el máximo heart_rate según la fórmula de Med Sci Sports Exerc.

```
age = int(input( Enter your age in years: ))
max heart rate = 206.9 - (0.67 age) # as per Med Sci Sports
Exerc.
target = 0.65 max heart rate
```

7.2 Entrada y Salida de ficheros

Los ficheros son tipicamente accedidos a través de la función built-in open(file_name,file_options). Estas opciones del fichero por defecto se corresponde a 'r', que consiste en sólo lectura. Otros métodos comunes son 'w', para abrir en modo escritura, 'a' para añadir texto al final del fichero. Es posible combinar opciones, como por ejemplo para tratar ficheros en binario con las opciones 'rb' o 'wb'.
Cuando se abre un fichero, la posición por defecto en modo 'r' y 'w' es el inicio del fichero. Es importante especificar que la diferencia entre 'w' y 'a' es que 'w' borra el contenido anterior del fichero y 'a' añade al final del fichero.
Cuando se termina de trabajar con un fichero es necesario cerrarlo con la función **fp.close()**.
Aquí se especifica un resumen de los métodos built-in que trabajan con ficheros.

Calling Syntax	Description
fp.read( )	Return the (remaining) contents of a readable file as a string.
fp.read(k)	Return the next k bytes of a readable file as a string.
fp.readline( )	Return (remainder of) the current line of a readable file as a string.
fp.readlines( )	Return all (remaining) lines of a readable file as a list of strings.
for line **in** fp:	Iterate all (remaining) lines of a readable file.
fp.seek(k)	Change the current position to be at the k^{th} byte of the file.
fp.tell( )	Return the current position, measured as byte-offset from the start.
fp.write(string)	Write given string at current position of the writable file.
fp.writelines(seq)	Write each of the strings of the given sequence at the current position of the writable file. This command does *not* insert any newlines, beyond those that are embedded in the strings.
print(..., file=fp)	Redirect output of print function to the file.

Leyendo un fichero

El comando básico para leer de un fichero es la función read(), llamada desde el objeto que maneja el fichero. Por ejemplo:

```
fp = open('input.txt','r')
fp.read() # para leer el contenido completo, devuelve un
string con todo el contenido
fp.readline() # para leer linea por linea, moviendo el cursor
a la siguiente linea en cada llamada
```

Escribiendo en un fichero

El comando write(string_to_write) escribe la string string_to_write en el sitio donde esté el cursor en el fichero. Si el fichero está en modo 'w' sobreescribirá el contenido del mismo, y si está en modo 'a' lo escribirá al final del fichero manteniendo el contenido previo.

```
fp = open('input.txt','a')
string_to_write='WHAT I WANT TO WRITE'
```

```
fp.write(string_to_write) # para escribir el contenido
string_to_write en el cursor donde esta el fichero
fp.writelines(seq) # para escribir el contenido de la
secuencia seq en el cursor donde esta el fichero en
diferentes líneas.
```

8 Manejo de excepciones

Las excepciones son eventos inesperados que ocurren durante la ejecución de un programa.
Una excepción se puede deber a un error lógico o a una situación que no se ha previsto.
Las excepciones en Python (como todo, por otra parte), son objetos que son lanzados (thrown) por el código que encuentra una circunstancia inesperada. Se deben tratar excepciones cuando se espera que un comportamiento anómalo pueda ocurrir (por ejemplo, la apertura de un fichero que no se encuentra).
Si no se tratan las excepciones, el programa para su ejecución y muestra un mensaje en consola.
En esta sección, se examinan los principales tipos de error en Python y el mecanismo para capturarlos y manejar los errores lanzados.

8.1 Tipos de excepción más comunes

Los tipos de excepción más comunes son los siguientes:

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Una de las excepciones más comunes es que un valor no pueda ser negativo, que un parámetro no sea de un determinado tipo esperado, que se acceda a una secuencia (lista, tupla o string) fuera de los índices válidos o que se intente acceder a conjuntos o diccionarios a un elemento no existente.

```
ValueError: invalid literal for int() with base 10: 'hello'
```

8.2 Lanzando (raising) excepciones

Una excepción se lanza cuando se ejecuta una sentencia raise con una infancia adecuada. Por ejemplo, si se desea calcular la raíz cuadrada de un número, este número no puede ser negativo. Se suele comprobar primero que los parámetros tienen el tipo adecuado, y luego que el valor del parámetro es válido.

```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError( x must be numeric )
```

```
    elif x < 0:
      raise ValueError( x cannot be negative )
    # do the real work here...
```

Las excepciones nacieron con la idea de prevenir todos los casos antes de que una excepción sea lanzada, aunque esto es a menudo imposible, se debe tratar de verificar todos los posibles casos y lanzar las excepciones adecuadas. Para ello, también es imprescindible el uso de pruebas unitarias.

Para ello, se pueden lanzar excepciones con try and except

```
def sqrt(x):
  try:
    math.sqrt(x)
  except TypeError:
    # do something else
  except ValueError:
    #do something else
```

Otro ejemplo de excepciones se puede producir cuando se está abriendo un fichero:

```
try:
  fp = open( sample.txt )
except IOError as e:
  print('Unable to open the file:', e)
```

Se pueden manejar diferentes excepciones a la vez, si se espera que puedan producirse diferentes errores y no se quieren tratar por separado.

```
try:
  fp = open( sample.txt )
except(ValueError, EOFError:
  print('Invalid File')
```

Si se desea que el programa no se pare y se continúe la ejecución ante determinado tipo de excepciones, se puede utilizar la sentencia **pass**.

```
except(ValueError, EOFError:
  pass
```

o bien pasar una excepción pero lanzar otra, en función del tipo:

```
age = -1 # an initially invalid choice
while age <= 0:
  try:
    age = int(input( Enter your age in years: ))
    if age <= 0:
      print( Your age must be positive )
  except ValueError:
    print( That is an invalid age specification )
  except EOFError:
```

```
    print( There was an unexpected error reading input. )
    raise # let s re-raise this exception
```

Se pueden capturar excepciones no esperadas con el tipo de excepción except.

9 Iteradores

En Python los contenedores básicos iterables son las strings, las listas, las tuplas, los sets (frozen y unfrozen) y diccionarios. Los tipos definidos por el usuario también pueden soportar la iteración. El mecanismo para iterar en Python sigue las convenciones:

- Un **iterador** es un objeto que maneja una iteración en una serie de valores. Si el objeto iter identifica un iterador, entonces cada llamada a la función next(iter) devuelve el siguiente elemento de la serie, con una excepción StopIteration cuando no hay más elementos.
- Un **objeto iterable** es un objeto que produce un iterador con la sintaxis iter(obj), siendo obj el objeto iterable.

Los iteradores no guardan su propia copia de una lista, sino que tienen un enlace al elemento correspondiente de la lista, por lo que los cambios en la lista afectan al iterador, tanto actualizaciones, borrados o inserciones.
Un ejemplo de un iterador

```
data = [1,2,3,4]
iter = iter(data)
while True:
  try:
   element = iter.next()
  except StopIteration:
    break
```

10 Documentación de Python

La importancia de la documentación en el código está fuera de toda duda. El creador de Python lo definió con una sencilla frase: “el código es muchas más veces leído que escrito”. Por ello, el código debe escribirse teniendo en cuenta dos audiencias, los usuarios y los desarrolladores. **Por ello, debe aportar información de qué hace cada función y cómo resuelve el problema resuelto.**

10.1 Comentarios vs Documentación

En general, los comentarios buscan describir el código por y para desarrolladores. Los comentarios ayudan en la lectura del código para entender el propósito y el diseño del mismo. Por resumir, el código te cuenta cómo estás resolviendo algo, y los comentarios deben especificar el por qué. Sin embargo, la documentación del código debe describir el uso y la funcionalidad de las funciones para los usuarios. **Tanto la documentación como los comentarios de código deben estar siempre en inglés.**

10.2 Comentarios

A continuación, ponemos un ejemplo de código Python donde se puede ver la sintaxis de varios comentarios.

```
def hello_world(param1):
  # Printing the param1
  print(param1)
```

No se deben utilizar comentarios de longitud excesiva (mayores de 72 caracteres, sino que en caso de ser largos se deben utilizar múltiples líneas para mantener el código limpio.

Los comentarios sirven para distintos propósitos, entre otros:

- **Revisar y plantear el código**: cuando se desarrolla nuevo código, puede ser apropiado dividirlo en diferentes pasos o etapas, aunque debe recordarse eliminar estos comentarios cuando se finaliza la implementación del código.

```
# first step
# second step
# third step
```

- **Describir el código**: se utilizan para explicar la intención de una sección de código.

```
# Attemp to establish a connection from the database
specificied in the param param3.
# If unsuccessful, prompt the error and a message
# to check the database status.
```

- **Describir algoritmos**: los comentarios son útiles para explicar cómo se ha desarrollado el algoritmo y por qué se ha implementado una determinada solución ante otras soluciones.

```
# Using bitonic sort for performance gains.
```

- **Etiquetas**: el uso de etiquetas es muy útil, por ejemplo, para etiquetar problemas o mejoras conocidas para implementar en el futuro.

```
# TODO: Still to implement unit testing to the function!
```

Los comentarios deben mantenerse concisos y concretos, así como estar lo más cerca posible del código que describen, no deben utilizar caracteres complejos ni incluir información redundante.

10.3 Documentación (Docstrings)

La documentación del código en Python se basa en **docstrings**. Se recomienda visitar las convenciones para la documentación de código en Python en el sitio web https://www.python.org/dev/peps/pep-0257/.

Un docstring es una cadena de caracteres (string) que está definido en la primera línea en un módulo, función clase o método. **Todos los módulos, funciones, clases y métodos públicos deben tener un docstring.** La documentación de un paquete se puede hacer en el fichero __init__.py in el directorio del paquete.

Los docstrings de una línea pueden ser definidos mediante cadenas de caracteres especificadas dentro de tres comillas dobles:

```
def documenting():
  """Example of one line docstring"""
  global documenting
  if documenting: return documenting
  ...
```

La documentación en una línea solo es apropiada en caso de funciones de C, donde la introspección no es posible. Como Python tiene un tipado dinámico, no se puede especificar en la función que tipo de datos va a devolver la función, por lo que esto debe ser mencionado en el docstring.

```
def documenting():
  """ Do X and return a tuple/list/whatever"""
```

Los docstrings multilínea consisten en una descripción más elaborada con el resumen de lo que realiza una clase/método/función/módulo en la primera línea seguido de una línea en blanco. Además, todos los docstrings deben finalizar con una línea en blanco (retorno de carro).

El docstring de un script debe ser utilizado como guía de utilización del mismo, especificando sus funciones, sus parámetros y la sintaxis de la línea de comando para llamar al script, así como qué ficheros utiliza, en caso de que utilice ficheros.

Los docstrings de paquetes deben contener un listado y resumen con los módulos que contiene el paquete.

Los docstrings de módulos deben contener un listado y resumen con las clases, excepciones y funciones que contienen el módulo.

Los docstrings deben ser útiles para usuarios sin experiencia (guía de utilización sencilla) y para usuarios con experiencia que quieran obtener o mejorar la eficiencia, para lo que se debe describir una completa descripción de todas las opciones y argumentos.

La función help(nombre_clase) imprime en consola el docstring definido.

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object.
 |  If encoding or errors are specified, then the object must
 |  expose a data buffer that will be decoded using the given
 |  encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if
 |  defined) or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 # Truncated for readability
```

Existen diferentes estilos y estándares de documentación según las características de los módulos o las corporaciones donde se desarrollan. Todos ellos comparten información común: **deben especificar qué realiza el módulo, qué realiza la función o método a comentar, qué parámetros recibe con el tipo correspondiente que acepta y qué devuelve dicha función junto con el tipo de dato retornado.** Por ejemplo, Google sacó unas reglas de estilo para los docstrings que se pueden consultar en:

https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html

La librería Numpy describe unos parámetros de estilo un poco diferentes:

https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html#example-numpy

Y a continuación podéis ver un ejemplo de un docString sobre una función hello_world(param1) puede verse a continuación:

```
def hello_world(param1):
  '''
  Hello world
  :param param1: number > 0
  :type reference: number
  :raises ValueError if param1 < 0
  Does it raise any other error? When?
  :returns nothing
  '''
  #print("Printing the type/class of param1",type(param1),
sep=" ")
  if param1 > 0:
    print("Printing the value of param1",param1, sep=" ")
    # Printing the type of param1
  else:
    raise ValueError("Ouch! Param1 is < 0 :(")
```

Aprovechamos aquí para recordar de nuevo el manejo de las excepciones y la diferencia entre el lanzamiento y la captura de excepciones.

Debemos distinguir muy bien el lanzamiento y la captura de excepciones.

Al **lanzar** una excepción indicamos que ha **ocurrido** un **evento inesperado**, y especificamos el tipo de evento inesperado que ha ocurrido con el tipo de excepción. Para ello, se utiliza la sintaxis raise. Todas las excepciones, como veremos en el tema 3, heredan de la clase Exception. De momento, debemos hacer un acto de fé para conocer lo que es la herencia y saber que las excepciones heredan de Exception.

Al **capturar** una excepción, por el contrario, lo que hacemos es definir qué acciones vamos a realizar si ocurre un evento inesperado (que se ha definido previamente en la función a la que se llama). Por ejemplo, si tenemos una función sqrt (raíz cuadrada), que **sólo** espera **números enteros positivos**, debemos definir en dicha función qué ocurre si recibimos **números enteros negativos**, como es el lanzamiento de ValueError, donde se le indica a aquél que haya llamado a dicha función que su valor no es apto, y qué ocurre si la función sqrt (raíz cuadrada) recibe un argumento que es cualquier otra cosa que no sea un número, en cuyo caso el error va a ser diferente, se va a producir un TypeError.

En la función o programa que llame a la función sqrt(x) debemos definir el flujo normal del programa (en el bloque try), y las acciones que vamos a realizar en caso de que ocurra cada uno de los errores.

Lanzar

```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be
numeric')
  elif x < 0:
    raise ValueError('x cannot be
negative')
  # do the real work here...
```

Capturar

```
def my_sqrt(x):
  try:
    math.sqrt(x)
  except TypeError:
    # do something else
  except ValueError:
    # do something else
```

Ejercicio: Define la función sqrt que utilice la librería

11 Testing

Una de las ventajas que ofrece la programación respecto a otras áreas de conocimiento es que se pueden ir probando todas las líneas de código, funciones, programas o partes de programas desarrollados.

En el caso de esta asignatura, no se va a profundizar en métodos de pruebas unitarias reales o con entornos integrados, pero sí se va a desarrollar una metodología trivial para probar todos los programas que realizamos. Para ello, **vamos a crear un main que va a contener la secuencia de operaciones y resultados esperados ante eventos esperados y, sobre todo, ante eventos inesperados.**

```
def hello_world(param1):
  '''
  Hello world
```

```
  :param param1: number > 0
  :type reference: number
  :raises ValueError if param1 < 0
  Does it raise any other error? When?
  :returns nothing
  '''
  #print("Printing the type/class of param1",type(param1),
sep=" ")
  if param1 > 0:
    print("Printing the value of param1",param1, sep=" ")
    # Printing the type of param1
  else:
    raise ValueError("Ouch! Param1 is < 0 :(")


def main():
  #Test 1. Expected right value
  try:
    hello_world(7)
    print("Test 1 Correct. Expected value is correct! We
are partially set")
  except ValueError as ve:
    print("ERROR Test 1. Check Value Error returned by the
function hello_word(param1)", str(e),
str(e.__class__.__name__), sep=" ")
  except Exception as e:
    print("ERROR Test 1. Any other Error returned by the
function hello_word(param1)", str(e),
str(e.__class__.__name__), sep=" ")

  #Test 2. Expected wrong value
  try:
    hello_world(-5)
    print("ERROR Test 2. Why damn am I here? Something
went wrong. Please check the function
hello_world(param1)")
  except ValueError as ve:
```

```
    print("Test 2 Correct. Unexpected is correct! We are
all set. We have tried unexpected events. Aren't we?")
  except Exception as e:
    print("ERROR Test 2Any other Error returned by the
function hello_word(param1)", str(e),
str(e.__class__.__name__), sep=" ")

  #Test 3. Unexpected value (any other thing than a
number?)
  try:
    hello_world("sdk")
    print("ERROR Test 2. Why damn am I here? Something
went wrong. Please check the function
hello_world(param1)")
  except ValueError as ve:
    print("ERROR Test 2. Why damn am I here? Something
went wrong, there is no value Error here. Please check the
function hello_world(param1)")
  except Exception as e:
    print("Test 3 Correct. Now we are ALL SET!! Any other
value than a number should return an Exception", str(e),
str(e.__class__.__name__), sep=" ")



if __name__ == "__main__":
  main()
```

Como se puede ver, el main prueba a ejecutar hello_world con el parámetro 7. Y no espera que ocurra una excepción, por ello el primer test (param1(7) es correcto. El segundo test comprueba qué ocurre al enviarle un valor negativo a la función param1 (param1(-5)). Al haber definido en la función el lanzamiento de una excepción de tipo valueError, esperamos dicho error (ver test 2) y lo lanzamos.

El test 3 comprueba qué ocurre ante eventos no esperados, como que el parámetro 1 sea cualquier otra cosa que un número. El test 3 comprueba que lanza una excepción general, que podría definirse como TypeError en lugar de la excepción general.

Como habréis podido observar, en cada una de las excepciones capturadas se puede definir un flujo de trabajo diferente. Por ejemplo, si quiero pedirle al usuario un número para calcular su raíz cuadrada y

posteriormente multiplicarlo por el número que el usuario elija, puedo hacer un bucle que se repita mientras el flujo de trabajo entre en alguna excepción y que salga cuando la secuencia de acciones vaya por el try. En caso de que ocurra alguna excepción se le volverá a pedir al usuario que introduzca un número por teclado, especificándole el motivo (número negativo o cualquier otra cosa que no sea un número) por el que debe introducirlo de nuevo.

Se recomienda realizar este ejercicio. Es muy completo ya que nos sirve para repasar muchos de los conceptos que vamos a utilizar durante la asignatura: estructuras de control de flujo, documentación, testing y manejo de excepciones.

12 Ejercicios

Todos los ejercicios deben contener las pruebas correspondientes que el estudiante considere necesarias. Todos los conceptos aquí repasados van a ser la base sobre la que vamos a trabajar durante la asignatura. El dominio de los tipos de datos, el control de flujo, la entrada/salida de datos por teclado/pantalla/ficheros, las excepciones y el testing (pruebas) es una condición necesaria para el correcto desarrollo del alumno. Todos estos conceptos, a excepción del testing, han sido ya estudiados en la asignatura de fundamentos de computadores de primer curso.

1. (**5 min**) Escribir un programa que defina variables que representen el número de días de un año, el número de horas que tiene un día, el número de minutos que tiene una hora y el número de segundos que tiene un minuto. A continuación, calcular el número de segundos que tiene un año y almacenar el valor del cálculo en otra variable.
2. (**10 min**) Calcular la suma de todos los múltiplos de 5 comprendidos entre 1 y 100. Calcular además cuantos hay y visualizar cada uno de ellos.
3. (**10 min**) Escribe un programa que calcule el mínimo y el máximo de una lista de números enteros positivos introducidos por el usuario. La lista finalizará cuando se introduzca un número negativo.
4. (**10 min**) Escribe un programa que visualice por pantalla la tabla de multiplicar de los 10 primeros números naturales.
5. (**10 min**) Escribe un programa que muestre por pantalla la lista de los 100 primeros números primos. Utilizar funciones para saber si un número es primo.
6. (**10 min**) Empleando una lista, escribir un programa que pida al usuario números enteros hasta que se introduzca el número 0. A continuación, calcular la media, el mínimo y el máximo de los datos introducidos. No utilizar las funciones built-in (max, min) proporcionadas por python. Crear las funciones que recorran las listas y calculen el máximo y mínimo y llamar a nuestras propias funciones.
7. (**10 min**) Escriba una función minmax(data) que tomando una secuencia de uno o más números como parámetro, devuelva el mínimo y el máximo en forma de tupla de longitud 2. No se pueden utilizar las funciones built-in max y min, sino que debemos implementarlas nosotros. Se deben utilizar todas las comprobaciones necesarias acerca de los tipos de datos.

8. (**10 min**) Escriba una función sumSquares(n) que devuelva la suma de todos los cuadrados de los números enteros positivos menores que n.
9. (**10 min**) Escribe un programa que acepte una cadena de caracteres (que podrá contener cualquier carácter a excepción del retorno de carro) y que diga cuántas vocales contiene. El programa debe tener en cuenta todos los casos de vocales (mayúsuclas, minúsculas y tildes). Se deben utilizar funciones.
10. (**10 min**) Escriba una función is_even(n) que tomando un valor entero, devuelva True si es par, y False en caso contrario. El programa debe lanzar excepciones si los tipos de datos enviados como parámetros no son enteros.
11. (**15 min**) Escribe una función de Python is_multiple(n,m), que tomando dos enteros, devuelva True si n es múltiplo de m, y False en caso contrario! Esto quiere decir que n=m*i siendo i algún número entero. El programa debe lanzar excepciones si los tipos de datos enviados como parámetros no son enteros. Escribe otra función divisors(n) que devuelva una lista con todos los números que son divisores de n
12. (**15 min**) Escribe un programa que lee una cadena de caracteres de teclado e indique si es o no palíndroma (se lee igual de izquierda a derecha que de derecha a izquierda, sin tener en cuenta los espacios en blanco y las mayúsculas). Por ejemplo: "dábale arroz a la zorra el abad".
13. (**15 min**) Escribe una función en Python que determine si todos los elementos de una lista son distintos o si hay elementos repetidos. En caso de que haya elementos repetidos, devolver True, en caso contrario False. Programe todas las excepciones necesarias.
14. (**20 min)** Python permite acceder a los diferentes elementos de una secuencia a través de su índice con la sentencia seq[k] con $0 < k < len(k) - 1$. Python también permite acceder a los elementos con índices negativos, siendo el índice -1 correspondiente con len(k) -1. Crear un programa que devuelva el índice k positivo que se corresponde con el índice j negativo. El programa sólo acepta como parámetros números enteros negativos menores que la longitud de la secuencia introducida.
15. (**25 min**) Escriba un programa que lea de un fichero csv (comma separated values) con dos filas diferentes cuyo título es edad y frecuencia_maxima que contienen la edad y la frecuencia máxima de diferentes pacientes. Los

valores de las columnas están separados por un ‘;’, y las filas por un retorno de carro. El programa debe calcular la frecuencia cardiaca media de los pacientes, la frecuencia cardiaca máxima y la frecuencia cardiaca media para pacientes de más de 31 años. Los resultados deben ser guardados en un fichero llamado ‘results.txt’.

16. (**25 min**) EXTRA. Escriba la descripción de una función que devuelva la lista reversa pasando como parámetro una lista L, con los elementos ordenados en orden inverso a como estaban. Tras esto, busca la implementación reverse() oficial de Python en Internet y compárala con la diseñada por ti.
17. (**45/55 min**) EXTRA: Escribir un programa que multiplique dos matrices. Sus dimensiones y valores deben de solicitarse al usuario por teclado y tras realizar la multiplicación debe visualizarse en pantalla ambas matrices y el resultado de la multiplicación.