Programación en Python

1. Introducción a los lenguajes de programación
2. *Datos y expresiones*
   - Números. Operaciones y expresiones. Variables. Booleanos. Strings. Uso de funciones matemáticas. Variables y paso de parámetros. Listas y tuplas.
3. Clases
4. Estructuras algorítmicas
5. Estructuras de Datos
6. Tratamiento de errores
7. Entrada/salida
8. Herencia y polimorfismo

2.1. Números

Python dispone de tres tipos de números con los que podemos trabajar

- `int`: números enteros
  - precisión ilimitada
  - aritmética *exacta*
- `float`: números reales
  - número real de unos 15 dígitos de precisión y rango ±1.7E+308
  - ocupa 64 bits (8bytes)
  - aritmética *aproximada* (errores de redondeo)
  - admite valores especiales como +∞, -∞ o nan (*not a number*)
- `complex`: números complejos
  - formados por dos números reales que actúan como parte real y parte imaginaria

Notas:

Aunque todos los números enteros están comprendidos dentro de los reales y éstos dentro de los complejos, existen diversos motivos para tener tipos de datos separados:

- *Enteros*: permiten una aritmética exacta y, como es lógico, no permiten manejar decimales.
  - Su tamaño en memoria depende del valor almacenado: cuanto más grande en valor absoluto, más memoria ocupan.
  - Su tiempo de cálculo depende también del tamaño.
  - Los usaremos para contar cosas. Por ejemplo el número de páginas de un libro o el número de personas en una sala.
- *Reales*: presentan errores de redondeo debido a que el número de decimales que pueden representar es limitado.
  - En total tienen unos 15 dígitos de precisión. Todo lo que pase de ahí, se descarta.
  - Por ejemplo, comprobar en el terminal de Python el resultado de este cálculo:

```
1-(5/3-4/3)*3
```

  - Usaremos los números reales para representar magnitudes físicas. Por ejemplo, la temperatura de una sala o la longitud de un ratón.
- *Complejos*: son más complicados de usar. Operar con ellos tarda más.
  - Los usaremos solo en caso necesario

Literales de los números

Permiten expresar valores

tipo	descripción	ejemplos

`int`	el número entero en decimal
	el número en octal, con `0o`
	el número en hexadecimal, con `0x`
	el número en binario, con `0b`
		`12  -37 1_000_000`
		`0o37`
		`0x2A`
		`0b100101`

`float`	el número con parte fraccionaria y/o
	
	notación exponencial con `e` o `E`
		`0.0  13.56  -12.0`
		
		`6.023E23 1.0e-6`

`complex`	se añade la letra `j` para indicar la parte imaginaria	`1 + 2j` `0.1 + 2.5j`

Notas:

Los literales de los números permiten expresar *valores concretos* en un programa Python.

Distinguimos el tipo de dato por el formato:

- *Enteros*: Para la notación habitual de números decimales se representa el número sin más.
  - Se pueden usar otras bases de numeración como octal (base 8), hexadecimal (base 16) o binario (base 2), pero esto es muy poco habitual.
  - Si el número es largo se puede usar el símbolo barra baja _ para separar los miles, millones, etc. Es recomendable hacerlo. Ejemplo: cien millones es `100_000_000`
- *Reales*: Siempre llevan parte decimal, aunque esta sea cero.
  - El carácter que se usa para separar la parte entera de la decimal es el *punto*, siguiendo la notación americana. Ejemplo: diez es `10.0`
  - Se puede usar notación exponencial para representar números de valor muy grande o de valor absoluto muy pequeño. En esta notación la letra E o e representa "*multiplicado por 10 elevado a la*".

    Por ejemplo, el número `1.02e6` representa $1{,}2 \cdot 10^{6}$ o `1.02e-6` representa $1{,}2 \cdot 10^{-6}$
- *Complejos*: Se representan como una suma de parte real e imaginaria (o solo parte imaginaria, en su caso), donde la parte imaginaria lleva la letra `j` inmediatamente a continuación del número, sin espacios y otros signos u operaciones.

Existen literales de otros tipos de datos, no numéricos, que veremos más adelante.

2.2. Operaciones y expresiones

Las *expresiones* permiten transformar datos para obtener un resultado

Se construyen con *operadores* y *operandos*

Operandos:

- constantes literales
- datos de los tipos predefinidos (atributos, variables, o argumentos de tipos predefinidos)
- funciones cuando retornan un valor de un tipo predefinido

Notas:

Ejemplo de una operación de suma:

a+7

Los datos en este caso son la variable a y el valor literal 7

Operaciones aritméticas

Principales operadores aritméticos: operan con números

Operación	Resultado
`x + y`	suma de x e y
`x - y`	diferencia x menos y
`x * y`	producto de x por y
`x / y`	cociente de x entre y; si los operandos son enteros, el resultado es real
`x // y`	cociente entero de x entre y; si los operandos son enteros, el resultado se trunca y es entero
`x % y`	módulo: resto de la división entera de x entre y
`-x`	cambio de signo de x
`abs(x)`	valor absoluto de x
`round(x)`	redondeo al entero más próximo
`x ** y`	x elevado a y

Notas:

Hemos visto arriba los principales operadores aritméticos, que son los que operan con números.

Entre ellos, hemos visto las tradicionales operaciones de suma, resta, multiplicación y división.

- La multiplicación usa el símbolo asterisco `*`, ya que el tradicional punto en medio de la línea (·) que se usa en la notación matemática no está disponible en los teclados normales.
- La división usa la barra de dividir en línea (`/`), pues las instrucciones del programa se escriben en línea.
  - Así, el cociente $\frac{a}{b}$ se escribe como `a/b`
  - Con objeto de no perder información, el resultado de una división de números enteros es un número real, aunque el cociente no tenga decimales. Así, `4/2` vale `2.0`

La división entera tiene dos alternativas si deseamos evitar la conversión a número real y seguir trabajando con números enteros:

- división entera: se usa el operador `//`. Si el resultado tuviese decimales, se trunca para obtener un número entero. Así, `5//3` vale `1`.
- resto entero :se usa el operador `%`, llamado módulo. El resultado es el resto de la división. Así, `5 % 3` vale `2`. El cociente entre `5` y `3` tiene un cociente entero de `1` y un resto de `2`.

Otras operaciones: valor absoluto de un número (`abs`) o redondear al entero más próximo (`round`).

Finalmente hemos visto el operador potencia o elevar a: `**`. Así, 4^3 se escribe como `4**3`

Precedencia

Cuando aparecen varios operadores en una expresión se evalúan en el orden marcado por su precedencia

- es el orden habitual de la notación matemática
- para la misma precedencia se usa orden de izquierda a derecha
- Ejemplo

4+5*2**2	$4 + 5 \cdot 2^2 = 24$

La precedencia se puede alterar con paréntesis (no `[]` ni `{}`)

(4+5)*2**2	$(4 + 5) \cdot 2^2 = 36$

Notas:

Salvo por el uso de símbolos diferentes, la notación matemática se parece mucho a la notación Python para las operaciones aritméticas.

También son iguales las reglas de precedencia. Cuando necesitamos expresar una precedencia diferente a la ordinaria, usamos paréntesis `()`.

Hay un caso donde tenemos que ser especialmente cuidadosos con los paréntesis: la división.

- Así, la operación

$$\frac{a+3}{z \cdot 8}$$

- requiere paréntesis tanto para el numerador como para el denominador, debido al uso de la división en línea: `(a+3)/(z*8)`

Tabla de precedencia

De mayor a menor

`**`	
`+, -, ~`	unario
`*, /, //, %`	
`+, -`	binario
`<<, >>`	
`&`	
`^`	
`|`	
`==, !=, >, >=, <, <=, is, is not, in, not in`	
`not`	
`and`	
`or`	

Notas:

La tabla de arriba muestra las reglas de precedencia para los principales operadores. Se incluyen operadores aritméticos y otros no aritméticos que veremos enseguida.

Quizás convenga explicar que los operadores + y - tienen dos modalidades:

- *unarios*: se aplican a un solo dato y representan:
  - la operación de identidad (+). Ejemplo +a, que es lo mismo que simplemente a
  - la operación de cambio de signo (-). Ejemplo: -a
- *binarios*: se aplican a dos datos y representan la suma y la resta.

Vemos en la tabla superior que la precedencia depende de si nos encontramos con el operador unario, o binario.

Promoción de tipos

Cuando se mezclan números de diferente tipo, los más "estrechos" se convierten automáticamente al tipo más "ancho"

El orden es:

- `int => float => complex`

Ejemplo:

- `5+4.5` da como resultado `9.5`
- `6+(7+8j)` da como resultado `(13+8j)`

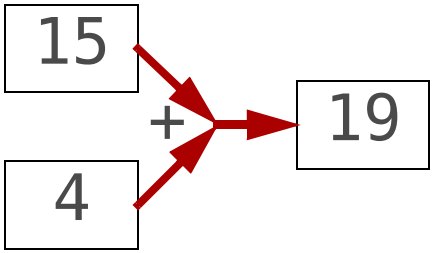
2.3. Variables

Los datos son casillas de memoria que contienen un *valor* de un *tipo* y se almacenan en la memoria en lugares llamados *objetos*

Habitualmente se crean con un literal o como resultado de una operación

15

15+4



Casi todos los objetos son *inmutables* (su valor no puede cambiar)

Es habitual referirse a un dato mediante una *variable*:

- *Etiqueta* que se refiere a un dato u objeto guardado en la memoria
- Su tipo es el del dato al que se refiere
- Internamente es una *referencia* al dato
  - información que indica en qué lugar de la memoria está el objeto

Notas:

Los datos de un programa Python se almacenan en casillas de memoria que llamamos *objetos.*

- Contienen un valor,
- de un tipo concreto.

En el ejemplo de la figura de arriba hay tres objetos:

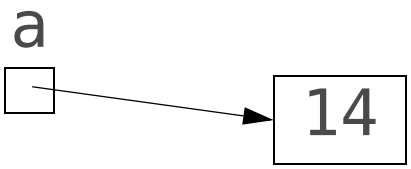
- El valor entero 15, que se ha obtenido con un literal.
- El valor entero 4, que se ha obtenido con un literal.
- El valor entero 19, que se ha obtenido sumando los dos literales anteriores.

Es habitual dar nombre a estos objetos. Este nombre es una etiqueta que llamamos *variable*.

Variables y asignación

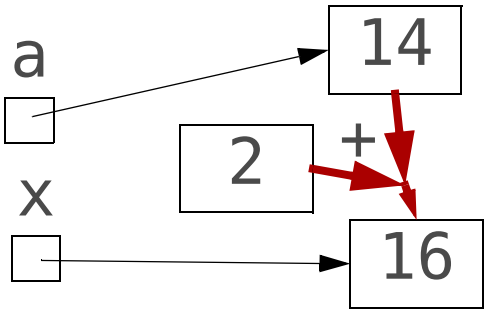
Para asignar un dato a una variable usamos el símbolo de asignación =

`a = 14`	asigna el dato 14 a la variable a si a no existe, se crea



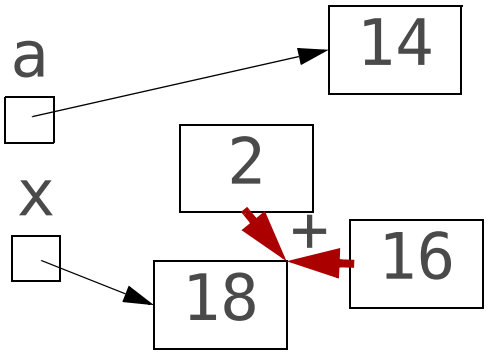
La variable se pone siempre a la izquierda

`x = a+2`	asigna el resultado de la suma (16) a la variable x



¡Cuidado! = no es una igualdad matemática

`x = x+2`	asigna el dato 18 a la variable x

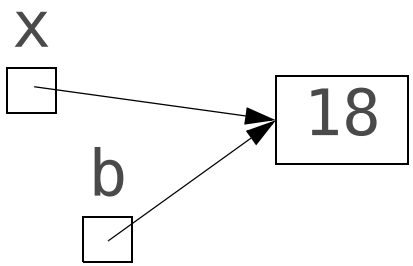


El código es secuencial: una variable (o cualquier nombre) no se puede usar antes de crearla

Asignación (cont.)

La asignación no crea objetos

`b = x`	asigna a la variable b el *mismo* dato (18) asignado a x



Notas:

Para crear una variable usamos la operación de asignación. Por ejemplo `x = 3` crea la variable `x` y hace que sea una etiqueta que se refiere al (apunta al) objeto que contiene el valor `3`.

Cuando la variable ya existe, si la volvemos a asignar se cambia la referencia. Por ejemplo, si ahora hacemos `x = 5`, la variable `x` pasa a referirse al objeto que vale `5`.

En la asignación siempre se pone:

- A la izquierda la variable.
- A la derecha un valor simple (como un literal u otra variable) o una expresión con operaciones.

Es muy importante acordarse de poner la variable a la izquierda del símbolo =.

Observar que, aunque se usa el mismo símbolo que en la operación matemática de igualdad, la asignación es completamente diferente.

- Por ejemplo, la expresión matemática `x=x+2` resultaría absurda, y sin embargo en Python es una operación de asignación completamente válida.

Un detalle a observar es que la asignación no crea objetos. Estos se crean mediante literales y como resultado de hacer operaciones con otros objetos.

Instrucciones de asignación

Hay instrucciones que combinan asignación y una operación aritmética

La asignación se hace después de calcular la expresión de la derecha

Instrucción	Resultado
`x = expr`	asigna a x la expresión de la derecha
`x += expr`	asigna a x la suma de x más la expresión de la derecha equivale a `x = x+expr`
`x -= expr`	asigna a x la resta de x menos la expresión de la derecha
`x *= expr`	asigna a x el producto de x por la expresión de la derecha
`x /= expr`	asigna a x el cociente de x entre la expresión de la derecha
`...`	hay operadores combinados con todos los operadores aritmé-ticos

Ejemplo que incrementa el valor asignado a x

```
x = x+1        o        x+=1
```

Tipos de variables por su ámbito

Variables *locales*

- definidas directamente en el interior de una función
- tienen una vida corta, solo mientras se ejecuta la función
  - aunque si la función es el programa principal (`main`) entonces su vida es la del programa, y por tanto larga
- se destruyen al acabar la función
- se usan solo para cálculos provisionales

Argumentos o parámetros

- definidos en los paréntesis del encabezamiento de la función
- se les da valor al invocar la función
- representan datos que la función obtiene del exterior
- su vida es corta, como las variables locales

Tipos de variables por su ámbito (cont.)

Variables *globales*

- definidas directamente en un módulo o fichero
- tiene una vida larga, igual a la del módulo
- su ámbito amplio puede provocar modificaciones *no intencionadas* por lo que están desaconsejadas; *debemos evitarlas, excepto si son constantes*

Entonces, ¿cómo creamos variables de vida larga?

Atributos

- definidas en el interior de las clases (que veremos más adelante)
- tienen una vida larga; están disponibles siempre que se necesiten
- su ámbito está restringido a la clase, lo que evita el peligro

Notas:

Hemos visto que según el lugar donde creamos las variables en el programa (según su ámbito), estas reciben diferentes nombres:

- Variables locales: definidas dentro de una función.
  - Servirán solo para almacenar cálculos intermedios, ya que desaparecen al finalizar la función.
- Variables globales: definidas directamente en el módulo (fuera de las funciones)
  - Las evitaremos por ser peligrosas, excepto si son constantes.
- Argumentos o parámetros: definidos dentro del paréntesis del encabezamiento de una función.
  - Sirven para pasar datos a la función desde el exterior.
  - También se destruyen al finalizar la función.
- Atributos: definidos en las clases.
  - Tienen vida larga.
  - Los veremos más adelante.

Según el ámbito, las variables pueden tener una vida corta (restringida a su ámbito) o larga (dura todo lo necesario).

- A veces nos interesan las variables de vida corta, pues hay datos intermedios que ya no necesitamos una vez usados, y así liberamos espacio en la memoria.
- A veces necesitamos variables de vida larga: las pondremos dentro de las clases, o en constantes, o como variables locales del programa principal `main()`.

Tipos de variables por su ámbito (cont.)

variable global
vida larga
¡peligrosa!

argumento
vida corta
existe solo mientras se ejecuta **func**
su valor viene de fuera de **func**

variable local
vida corta
existe solo mientras se ejecuta **func**
su valor viene del interior de **func**

```
# -*- coding: utf-8 -*-
"""
Programa que ilustra ámbitos de variables

@author: Michael González
@date  : 17/ene/2019
"""

g: float = 0.0

def func(x: float):
    """
    Muestra en pantalla x+1
    """

    y: float = 1.0
    print(x+y)

def main():
    """
    Programa principal
    """
    func(12.0)
```

Notas:

En el ejemplo que se muestra, se crean tres variables: `g`, `x` e `y`.

Podemos observar que, al crearlas, hemos hecho una anotación de tipo con el formato:

```
variable : tipo = valor
```

La anotación de tipo no es obligatoria en Python (podríamos escribir simplemente `variable = valor`), pero nosotros intentaremos hacerla siempre pues aclara al lector el tipo de la variable que se quiere crear.

Esta anotación de tipo la hacemos solo al crear la variable. Posteriormente, durante su uso normal o incluso aunque volvamos a asignarle un nuevo valor, ya no se pone.

Cada variable tiene su ámbito:

- Las variables `x` e `y` pertenecen a la función `func()`.
  - Las usa solo la función `func()` y no se pueden usar desde la función `main()`.
- La variable `g` es global, al estar definida fuera de las funciones.
  - Siempre intentaremos evitar estas variables, a no ser que sean *constantes*. Las veremos más adelante.

Por cierto, evitaremos en lo posible nombres cortos como `g`, `x` e `y`. Intentaremos utilizar nombres más largos que permitan describir el dato que contienen. Aquí hemos usado un nombre corto porque el programa que se muestra es solo un ejemplo que no hace nada útil.

Ejemplo de un programa con datos

Cálculo de la media de tres notas

- las notas (variables enteras) deben tener vida larga
- como aún no hemos visto las clases serán variables del `main()`

El módulo tendrá:

- función `media_entera`:()
  - recibe las tres notas como parámetros y retorna la media *entera*
- función `media_real()`:
  - recibe las tres notas como parámetros y retorna la media *real*
- una función `main()` con tres variables enteras (las tres notas)

Podríamos hacer todos los cálculos en el `main`, sin otras funciones

- pero es conveniente aplicar el principio de "*divide y vencerás*"
- dividiendo la funcionalidad en varias partes

Notas:

En este ejemplo manejaremos tres notas de un alumno y haremos diversos cálculos con ellas.

Queremos que estas tres notas tengan vida larga:

- Ya sabemos que no queremos usar variables globales.
- La mejor solución sería usar atributos de una clase, pero veremos las clases más adelante.
- Para salir del paso, las notas serán *variables locales* de la función principal `main()`.
  - Aunque están restringidas a su uso solo en la función `main()`, como el programa se termina al finalizar la función `main()`, esta es la única función cuyo ámbito tiene una vida igual a la del programa.
  - Es decir, las variables locales del `main()` tienen vida larga.

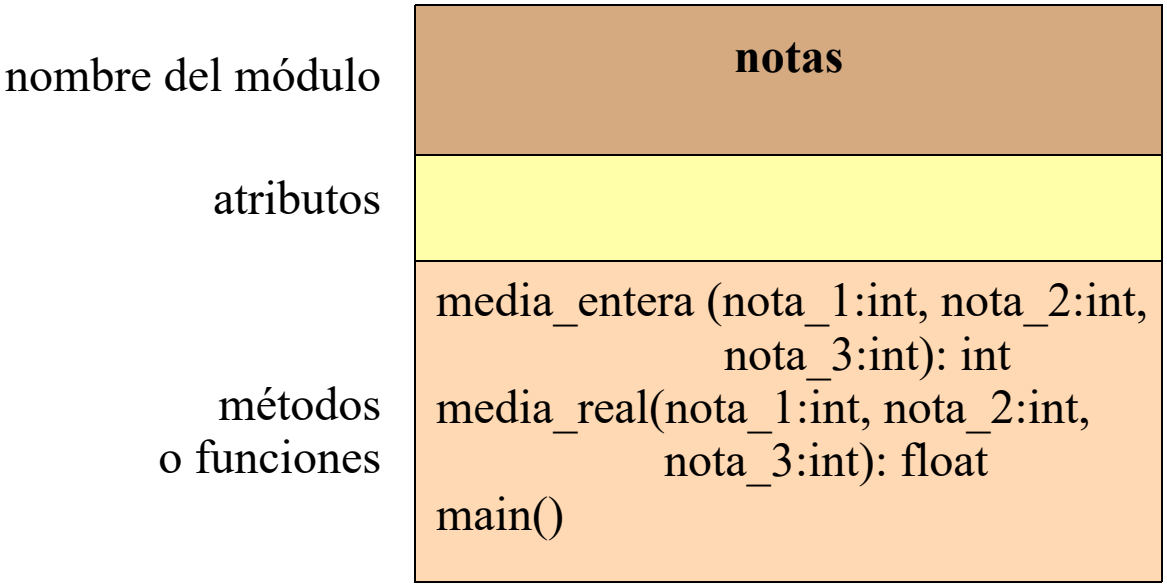
Entonces ¿cómo se pueden usar esos datos (las notas) desde las otras funciones que queremos escribir?

- Les pasaremos los datos mediante *parámetros*.

Las funciones que queremos escribir hacen cálculos con sus datos, obtienen un *resultado* y lo retornan con la instrucción `return` para que el `main()`, que es desde donde se invocan, pueda usarlo.

Diagrama de clases

Podemos interpretar el módulo como una clase



Observar los encabezamientos de funciones:

- formato de los argumentos: `nombre: tipo`
- valor retornado (en su caso): `encabezamiento: tipo`

Notas:

Aunque veremos las clases más adelante, a veces podemos interpretar un módulo Python como si fuera una clase, ya que tiene algunos de sus elementos (aunque con otra estructura y propiedades).

En una clase encontraremos:

- Nombre de la clase
- Atributos, o datos de la clase
- Funciones, también llamados métodos.

Estos elementos se suelen describir mediante un diagrama de clase similar al que se muestra en la figura superior.

- Es un rectángulo con tres partes, para cada grupo de elementos.

En nuestro caso, el módulo no tiene datos globales. Tiene:

- Nombre del módulo
- Funciones.

Para describir las funciones en el diagrama se pone algo parecido a su encabezamiento en Python, pero con el formato ligeramente distinto que se muestra en la figura. Observar:

- La ausencia de la instrucción **`def`**.
- El uso de "`:`" en lugar de la flecha "`->`" para anotar el tipo del dato retornado por la función.

Ejemplo (cont)

```
# -*- coding: utf-8 -*-
"""
Opera con las notas de tres alumnos

@author: Michael
@date  : ene-2019
"""
```

Ejemplo (cont)

```
def media_entera(nota_1: int, nota_2: int,
                 nota_3: int)-> int:
    """
    Calcula la media de tres notas como numero entero

    Args:
        nota_1: nota del primer examen
        nota_2: nota del segundo examen
        nota_3: nota del tercer examen
    Returns:
        la media entera de las tres notas

    """

    return (nota_1+nota_2+nota_3)//3
```

Ejemplo (cont)

```
def media_real(nota_1: int, nota_2: int,
               nota_3: int)-> float:
    """
    Calcula la media de tres notas como numero real

    Args:
        nota_1:  nota del primer examen
        nota_2:  nota del segundo examen
        nota_3:  nota del tercer examen
    Returns:
        la media real de las tres notas
    """

    return (nota_1+nota_2+nota_3)/3
```

Ejemplo (cont)

```
def main():
    """
    Programa principal

    Contiene tres notas de un alumno y muestra su media
    de dos formas
    """

    # Declaramos las notas y les damos valor
    nota_1: int = 7
    nota_2: int = 8
    nota_3: int = 8

    # Mostramos resultados
    media: int = media_entera(nota_1, nota_2, nota_3)
    print("Nota media entera =", media)

    # Esta vez sin usar una variable intermedia
    print("Nota media real   =",
          media_real(nota_1, nota_2, nota_3))
```

Notas:

Como podemos ver en el código, las funciones hacen un cálculo del valor medio sumando las tres notas y dividiendo por tres, y retornan el resultado de ese cálculo para que el `main()` lo pueda usar.

- la diferencia es que una opera con la división entera (`//`) y la otra con la división real (`/`).

Lo que hace la función main() es:

- Crea tres variables para las tres notas y les asigna valor. Se usan anotaciones de tipos.
- Invoca a la función `media_entera()` pasándoles como parámetros las tres notas.
  - Observar que al invocar a la función ya no se utilizan anotaciones de tipo. Solo se hizo al crear las funciones.
- Crea una variable llamada `media` y le asigna el valor retornado tras invocar a la función `media_entera()`. Como aquí estamos creando una variable nueva, sí ponemos la anotación de tipo.
- Muestra en pantalla con la instrucción `print()` el resultado almacenado en la variable `media`.
- Finalmente hace un proceso parecido con la media real, pero en este caso, a modo ilustrativo, no guarda el resultado de invocar a `media_real()` en una variable, sino que lo muestra directamente en pantalla con la instrucción `print()`.

Orden de ejecución

Es importante darse cuenta que el orden de ejecución lo definen las instrucciones del programa principal, `main()`

- A pesar de que las funciones `media_entera()` y `media_real()` están escritas antes que el `main()`, sus instrucciones solo se ejecutan cuando el `main()` las invoca

Documentación de funciones

Los *docstrings* que documentan las funciones tienen estos requisitos:

- Mostrar una descripción breve de lo que hace la función
  - No cómo está hecha. Esto se hace en los comentarios internos
- En su caso, mostrar una descripción de lo que significa cada argumento, y de sus unidades si las tiene
- En su caso, mostrar una descripción de lo que retorna la función y de sus unidades si las tiene

En el ejemplo se muestra el estilo de documentación de Google

- Hay otros estilos frecuentes

Otros comentarios sobre el ejemplo

- argumentos de un método: datos del exterior, que el método necesita
- instrucción `return`, para expresar el valor de retorno de un método (su respuesta)
- operador de asignación: "="
- operador de suma aritmética: "+"
- `print()` con varios parámetros, separados por comas
- uso de paréntesis
- divisiones reales y enteras
- comentarios de documentación e internos
- anotaciones de tipo
  - en las variables: `variable: tipo`
  - en el valor de retorno de la función: `encabezamiento-> tipo:`

Comentarios de documentación e internos

Otros comentarios internos

- ayudan a entender el código
- formato: texto entre # y el final de la línea
- se suelen poner encima de la instrucción o instrucciones a las que afecta

```
# Esto es un comentario
instrucciones a las que afecta
```

Constantes

Algunos lenguajes disponen de constantes, similares a las variables pero a las que no se puede cambiar el dato asignado

En Python no hay constantes, pero para indicar al programador que se *desea* que a una variable no pueda asignársele un valor nuevo se recurre a una convención

- se pone su nombre en mayúsculas
- Ejemplo:

```
CTE_DE_PLANK: float = 6.626070150E-34 # J*s
```

Se pueden definir como variables globales, ya que (se supone que) nadie les cambiará su valor

Es muy importante usar constantes para mejorar la legibilidad, para cualquier valor literal que preveamos que podría cambiar, o para cualquier valor literal que se repita

Notas:

Usaremos constantes *globales* cuando deban ser compartidas por varias funciones.

Cuando solo se usen en una función es mejor definirlas como constantes *locales* a la función, para ahorrar memoria.

Importancia del uso de constantes:

- Las constantes nos permiten dar nombre a un valor, para mejor legibilidad de un programa.
  - Por ejemplo, es mucho más fácil de entender BYTES_POR_SEGUNDO, en un programa de comunicaciones, que un simple número como 1_000_000.
- Cuando hay valores que preveamos que podrían cambiar para configurar el programa de otra forma, es conveniente tenerlos agrupados al principio del programa, en un lugar bien visible. Por ejemplo:
  - máximo aforo de personas (esto podría cambiar en el futuro si se cambian las condiciones de una sala)
  - longitud de mi coche (esto podría cambiar en el futuro si cambio de coche)
  - número de ejes del robot (esto podría cambiar si modificamos el robot y le añadimos más capacidad de maniobra).
- Cuando hay valores literales que se repiten dos o más veces, para evitar esta repetición. Esto facilitará un cambio futuro del programa evitando inconsistencias.

Constantes en Python 3.8

Además, a partir de Python 3.8 es posible indicar de una forma más explícita que deseamos que una variable tenga un valor constante

Para ello, se pone una anotación de tipo con el formato `Final[tipo]`

- Que indica que la variable toma su valor final

Para poder usar la anotación `Final`, es preciso poner al principio del módulo una instrucción de tipo **`import`**, como en el ejemplo:

```
from typing import Final

GRAVITACION_UNIV: Final[float] = 6.674e-11 #N*m**2*kg**-2
```

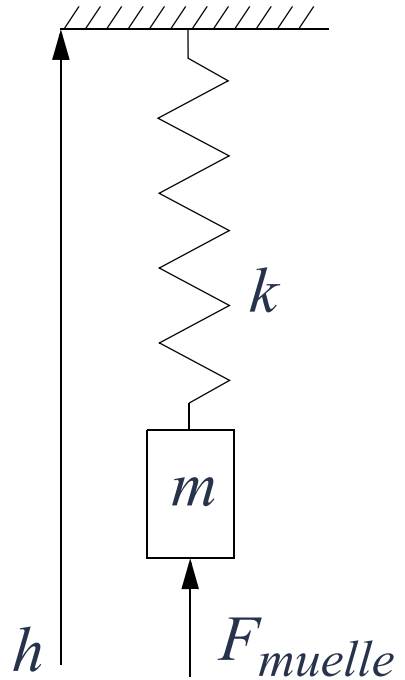
Intentaremos usar esta anotación de tipos siempre que se pueda

Ejemplo

Cálculo de la trayectoria de una masa suspendida de un muelle

Usaremos la ley de Hooke que nos da la fuerza del muelle en función de la altura (h)

$$F_{muelle} = -k \cdot h$$



Notas:

En este ejemplo crearemos un simulador que nos permite calcular en diversos momentos la posición (o altura) y velocidad de la masa suspendida del muelle.

Como sabemos las fuerzas que actúan sobre la masa, que son la fuerza del muelle y la gravedad, podemos sumarlas para calcular en cada momento la aceleración.

- Si quisiéramos tener en cuenta el rozamiento, podríamos sumar también las fuerzas del rozamiento interno del muelle y el rozamiento con el aire.

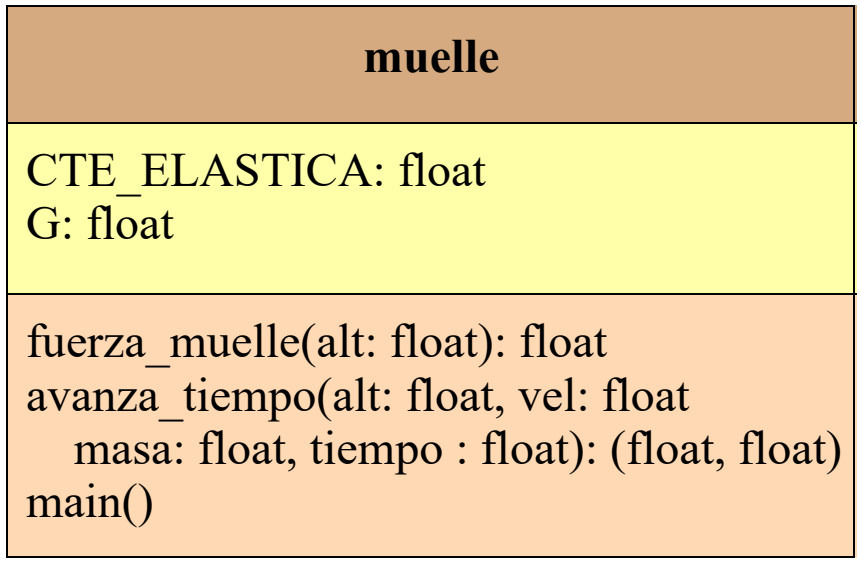
A partir de la velocidad y la aceleración podemos calcular la posición y nueva velocidad, aplicando las conocidas ecuaciones del movimiento uniformemente acelerado.

- La aceleración es variable, pues depende de la altura.
- Sin embargo, a efectos de la simulación supondremos que para intervalos de tiempo pequeños la aceleración apenas varía en ese intervalo.
- Ello nos permite aplicar las ecuaciones de aceleración constante.
- Cometeremos un error muy pequeño, casi despreciable, siempre que el intervalo de tiempo sea muy pequeño.

Diseño del módulo:

Diagrama de clases

- constantes para la constante del muelle y la gravedad
- una función que devuelve la fuerza del muelle, dada la altura
- una función que calcula la *nueva* altura y velocidad, dados los valores *actuales* de altura y velocidad, la masa y un intervalo de tiempo
  - retorna dos valores (una tupla)
- una función `main`



Una *tupla* es una secuencia inmutable de elementos. Ejemplo

```
mi_tupla = (1, 2, 3, "juan")
```

Notas:

En este diseño hay dos datos constantes, que ponemos como constantes globales.

Además, hay dos funciones y un `main()` que contiene los datos de altura y velocidad actuales y organiza la ejecución.

Puesto que la función `avanza_tiempo()` debe retornar dos valores, la altura y velocidad nuevas, y estrictamente una función Python solo puede retornar un valor, utilizamos la estructura de datos llamada tupla.

- Una tupla es una secuencia de elementos puestos entre paréntesis y separados por comas.
- En nuestro ejemplo usaremos una tupla de dos números reales, que representan la altura y velocidad. Por ejemplo, podrían ser (4.5, 7.9).

Ejemplo (cont.)

```
# -*- coding: utf-8 -*-
"""
Simula el movimiento oscilatorio de una masa que
pende de un muelle, sin considerar rozamientos

Se usan unidades del sistema internacional:
    kg para la masa
    m para la altura

El cero de altura se considera en el punto en el que el muelle
está en reposo. El sentido de la fuerza y altura es positiva
hacia arriba. Los datos principales son:
    alt   : altura actual
    vel   : velocidad actual de la masa, en m/s
            inicialmente es cero
    masa  : masa del objeto
    tiempo: tiempo transcurrido

@author: Michael
@date  : ene-2019
"""
```

Ejemplo (cont.)

```
from typing import Tuple
from typing import Final

#Constantes:
#    CTE_ELASTICA: Constante elástica del muelle, en N/m
#    G: Gravedad terrestre, en m/s**2

CTE_ELASTICA: Final[float] = 0.12

G: Final[float] = 9.8
```

Ejemplo (cont.)

```
def fuerza_muelle(alt: float)-> float:
    """
    Calcula la fuerza que ejerce el muelle
    dada la altura de su extremo

    Args:
        alt: la altura del extremo del muelle, en m
    Returns:
        la fuerza que ejerce el muelle, en N
    """

    # retornamos la fuerza calculada con la ley de Hooke
    return -CTE_ELASTICA*alt
```

Ejemplo (cont.)

```
def avanza_tiempo(alt: float, vel: float, masa: float,
                  tiempo: float)-> Tuple[float, float]:
    """
    Calcula la nueva altura y velocidad de la masa,
    transcurrido el tiempo especificado, que debe ser pequeño

    Args:
        alt: la altura actual del extremo del muelle, en m
        vel: la velocidad actual del extremo del muelle, m/s
        masa: la masa del cuerpo que pende del muelle, en Kg
        tiempo: el tiempo transcurrido, en s
    Returns:
        la nueva altura y velocidad de la masa (m y m/s)
    """
    # Calculamos la fuerza aplicada sobre la masa, en N
    fuerza: float = fuerza_muelle(alt)-masa*G
    # obtenemos la aceleración por la ley de Newton
    aceleracion: float = fuerza/masa
    # ecuaciones del movimiento uniformemente acelerado
    return (alt+vel*tiempo+(aceleracion*tiempo**2)/2,
            vel+aceleracion*tiempo)
```

Ejemplo (cont.)

```
def main():
    """
    Simula el movimiento de una masa suspendida de un
    muelle durante un tiempo, y pone en pantalla
    altura y velocidad. Unidades del S.I
    """

    masa: float = 0.25
    incremento: float = 0.01
    alt: float = 0.2
    vel: float = 0
    # Avanzar la simulación
    (alt, vel) = avanza_tiempo(alt, vel, masa, incremento)

    # mostrar resultados con un f-string

    print(f"Alt= {alt} m. Vel= {vel} m/s")
```

observar la 'f'

Se reemplaza por el valor de `alt`

Se reemplaza por el valor de `vel`

Notas:

En este ejemplo:

- `fuerza_muelle()` es una función trivial.
- `avanza_tiempo()` es una función que calcula la fuerza del muelle, a partir de ella la aceleración a la que está sometida la masa, y retorna una tupla con dos valores: las nuevas altura y velocidad. Estos valores se calculan con la fórmulas del movimiento uniformemente acelerado.
- `main()` es el programa principal. Crea las variables con los datos iniciales. Luego avanza la simulación invocando a la función `avanza_tiempo()`. Observar que se usa un intervalo temporal pequeño, ya que de lo contrario la aproximación de aceleración constante no sería válida. Los dos valores retornados por la función, que son las nuevas altura y velocidad, se guardan de nuevo en las variables `alt` y `vel`, sobreescribiendo sus valores anteriores. Finalmente, pone en pantalla los resultados obtenidos, usando una instrucción `print()` y un *f-string*.
  - Para una simulación realmente útil, habría que llamar muchas veces a la función `avanza_tiempo()`, e ir mostrando los sucesivos resultados. En los ejemplos de clase hay un programa que hace esto (`main3()`) y muestra una gráfica de la trayectoria de la masa.

Un *f-string* es un texto formateado. Se crea poniendo un texto entre comillas y anteponiendo la letra `f`. Dentro del texto, se ponen plantillas para mostrar datos, encerradas entre llaves `{}`. Dentro de la plantilla se pone una variable o un cálculo, cuyo valor reemplazará la plantilla en el texto.

- Si, por ejemplo, `alt` vale `4.5` y `vel` vale `7.9`, estos valores reemplazarán a sus plantillas dentro del *f-string* `f"Alt= {alt} m. Vel= {vel} m/s"` y obtendremos el texto `"Alt= 4.5 m. Vel= 7.9 m/s"`.
- Los *f-strings* nos permiten mostrar uno o varios resultados con texto alrededor.

A observar en este ejemplo:

- creación de constantes, expresadas directamente en el módulo
- creación de variables locales
- uso de *f-strings* (o strings con formato) para mostrar varios datos
  - delante del string se pone una `f` o `F`
  - las variables a mostrar en mitad del texto se encierran entre `{}`
- uso de una tupla con dos valores
- comentarios de documentación con:
  - explicación general
  - explicación de los parámetros con sus unidades
  - explicación del valor retornado con sus unidades

2.4 Booleanos

El tipo `bool` permite representar los valores lógicos verdad (`True`) o falso (`False`)

- también llamamos a este tipo de datos *booleano*

Es extremadamente importante, pues casi todos los programas deben tomar decisiones, que habitualmente se basan en valores lógicos

Una forma habitual de obtener valores booleanos es mediante los operadores de comparación o relacionales

- el resultado de la comparación es un booleano

Notas:

Una de las capacidades más importantes del computador es la toma de *decisiones*. Muchas de las decisiones se basan en *valores lógicos*. Por ejemplo, podremos decidir hacer una cosa si una condición es *cierta*, u otra cosa si es *falsa*.

Los valores cierto y falso o verdad y falso se representan mediante el tipo de datos booleano (tipo `bool`), que solo puede tomar dos valores: `True` o `False`. Estos valores son los literales del tipo.

Veremos a continuación los *operadores relacionales*, que nos permitirán obtener valores booleanos con facilidad.

Operadores relacionales

Operación	Resultado (`True` si se cumple la condición, `False` en caso contrario)
`x == y`	x es igual a y
`x != y`	x es distinto de y
`x > y`	x es mayor que y (estrictamente mayor)
`x >= y`	x es mayor o igual que y
`x < y`	x es menor que y (estrictamente menor)
`x <= y`	x es menor o igual que y
`x is y`	x e y están asignadas al mismo objeto
`x is not y`	x e y no están asignadas al mismo objeto

Notas:

Los operadores relacionales permiten comparar dos valores para obtener un resultado booleano, verdad o falso.

Los valores que podemos comparar pueden ser números, textos y muchos otros tipos de datos.

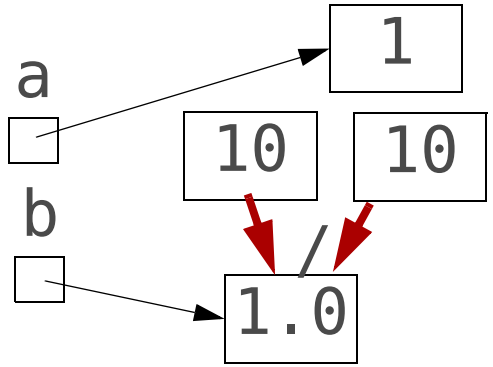
Los operadores relacionales presentan similitudes con los operadores matemáticos equivalentes, pero hay diferencias:

- El operador de igualdad en matemáticas es el símbolo `=`. Sin embargo, ya sabemos que en Python ese símbolo se usa para otra instrucción, que es la asignación. Por ello hay que usar uno diferente, en este caso dos símbolos igual seguidos: `==`
- En el teclado del computador no existen símbolos matemáticos como ≠ ≥ ≤. Por ello se utilizan secuencias de dos caracteres, que los imitan. Respectivamente, `!=` `>=` `<=`

Existen dos operadores (**`is`** e **`is not`**) para comprobar si dos variables se refieren o no al mismo objeto. Así, al ejecutar este fragmento de código:

```
a=1
b=10/10
print(a==b, a is b)
```

Se obtiene `True False`, pues `a` y `b` apuntan a valores idénticos (`1` y `1.0`), pero no al mismo objeto, ya que uno es un número entero y otro un real.



Operadores lógicos

Podemos trabajar con los operadores lógicos para escribir expresiones lógicas a partir de datos booleanos

Operación	Resultado (`True` si se cumple la condición, `False` en caso contrario)
x **`or`** y	o lógico: si x es False, entonces y, si no, x
x **`and`** y	y lógico: si x es False, entonces x, si no, y
`not` x	negación: si x es False, entonces True, si no, False

- por eficiencia, las operaciones **`or`** y **`and`** son condicionales:
  - **`or`**: el elemento derecho solo se evalúa si el izquierdo es `False`
  - **`and`**: el elemento derecho solo se evalúa si el izquierdo es `True`
- **`not`** tiene menor precedencia que los operadores no booleanos
  - **`not`** `a == b` se interpreta como **`not`** `(a == b)`
  - `a ==` **`not`** `b` es un error de sintaxis

Notas:

En ocasiones tenemos que trabajar con dos o más valores booleanos para obtener otro booleano. Esto lo hacemos con los operadores lógicos.

Los operadores lógicos están presentes en el lenguaje matemático y también en el lenguaje natural. Por ejemplo:

- Si llueve **y** hace frío me pongo el chubasquero.
- Si tengo hambre **o** sed me voy a una cafetería.

En Python podríamos escribir estas expresiones relacionales, suponiendo que `llueve`, `hace_frio`, `tengo_hambre` y `tengo_sed` son variables booleanas ya creadas:

Nombre	Tipo	Tamaño	Valor
hace_frio	bool	1	False
llueve	bool	1	True
me_pongo_el_chubasquero	bool	1	False
tengo_hambre	bool	1	True
tengo_sed	bool	1	False
voy_a_una_cafeteria	bool	1	True

Explorador de variab... | Ay... | Análisis del cód... | Gráfi... | A

```
me_pongo_el_chubasquero: bool = llueve and hace_frio
voy_a_una_cafeteria: bool = tengo_hambre or tengo_sed
```

Tema aparte es pensar si estas expresiones son apropiadas. Por ejemplo, tal como está escrito, si llueve pero no hace frío, no me pondré el chubasquero.

Ejemplo con booleanos

Escribiremos un programa para determinar si un año entre el 2000 y 2099 cuyo valor introducimos por teclado es bisiesto o no

- el año es bisiesto si es múltiplo de 4

Para determinar si un número es múltiplo de otro usaremos el operador módulo (%)

- si el resto de a/b es cero, a es múltiplo de b

```
a%b == 0
```

Ejemplo

```
# -*- coding: utf-8 -*-
"""
Programa que determina si un año es bisiesto

Contiene una función para determinar si un año es bisiesto
y un main que lee el año del teclado y muestra el resultado

@author: Michael
@date  : ene 2019
"""
```

Ejemplo (cont.)

```
def es_bisiesto(year: int)-> bool:
    """
    Determina si un año entre el 2000 y el 2099 es bisiesto

    Para otros años hay que usar reglas más complejas

    Args:
        year: el año

    Returns:
        un booleano que indica si el año es bisiesto o no
    """

    return year%4 == 0
```

Ejemplo (cont.)

```
def main():
    """
    Programa que lee un año de teclado y
    muestra en pantalla si es bisiesto o no
    """

    # Lee el año del teclado
    year: int = int(input("Introduce el año:"))

    # Determina si es bisiesto o no
    bisiesto: bool = es_bisiesto(year)

    # Muestra el resultado en la pantalla con f-strings
    print(f"El año {year} es bisiesto: {bisiesto}")
```

A observar en el ejemplo

- uso de expresiones relacionales
- uso de variables booleanas
- lectura del teclado
  - con `input()` leemos un string (texto)
  - con `int(texto)` convertimos el texto a entero
  - se podría hacer con `float(texto)` para convertir a real

NOTAS:

En este ejemplo introducimos por primera vez la lectura de datos por teclado, además de trabajar con booleanos y expresiones relacionales.

La función `es_bisiesto()` recibe como parámetro un número entero que contiene un año de este siglo. Calcula mediante una operación módulo con `4` el resto de la división. Sabemos que si este resto es cero, el año es divisible entre `4` y por tanto es bisiesto. Si el resto no es cero, el año no es bisiesto. Obtendremos un booleano que indica si el año es bisiesto o no mediante una expresión relacional. Finalmente, la función retorna ese booleano para ser usado en el lugar del programa desde donde se invoca a esta función.

EL programa principal, `main()`, lee de teclado el año, mediante la instrucción `input()`. Esta función recibe como parámetro un texto que se pone en la pantalla para explicarle al usuario qué tiene que teclear. La función retorna un texto. Como nosotros necesitamos un número entero, realizaremos una conversión de tipo mediante la expresión:

```
int(texto)
```

Una vez hecha la conversión a entero, guardamos el año en una variable llamada `year`.

Posteriormente, invocaremos a la función `es_bisiesto()` y obtendremos el valor retornado por esta, que es un booleano. Mostraremos en pantalla el año y el booleano, mediante un *f-string*, como ya vimos en un ejemplo anterior.

2.5 Strings (Textos)

El tipo de datos str permite almacenar secuencias de cero o más caracteres

Los literales de string se forman poniendo texto entre comillas

Strings unilínea	`'Un texto'`
	`"otro texto"`
Strings multilínea	`'''otro más'''`
	`"""y otro más"""`

Operador de concatenación: crea un nuevo string a partir de otros dos,

```
"Ana"+" está aquí"  # resultado: "Ana está aquí"
x = 8.5
"Valor ="+x  # incorrecto; usar "Valor ="+str(x)
```

Los strings son *inmutables*

Notas:

Además de los números y booleanos, es muy habitual la necesidad de almacenar y manipular textos.

El tipo de dato `str` permite guardar textos de longitud variable y es uno de los tipos más habituales en muchos programas.

- De hecho, ya lo hemos utilizado en ejemplos anteriores siempre que ponemos en el programa un texto entre comillas.

Podemos convertir datos de otros tipos a texto, con la conversión:

```
str(valor)
```

Mediante el operador + podemos unir (concatenar) dos textos.

Al igual que la mayoría de los tipos de datos en Python, los strings son inmutables.

- Cuando manipulamos textos, por ejemplo mediante la concatenación, los textos originales no cambian. Se crean nuevos objetos de tipo texto con el valor nuevo. Así, `"mi"+"casa"` produce un nuevo objeto de tipo `str` apuntando al objeto `"micasa"`.

Ejemplo de programa con variables de texto

```
# -*- coding: utf-8 -*-
"""
Programa para trabajar con el nombre de una persona y el de su
padre

@author: Michael
@date  : ene 2019
"""

def main():
    """
    Programa que pregunta dos nombres por teclado y
    los muestra en pantalla concatenados
    """

    nombre: str = input("Introduce tu nombre: ")
    padre: str = input("El nombre de tu padre: ")

    print(f"El padre de {nombre} es {padre}")
```

Notas:

El ejemplo anterior pide dos textos por teclado, usando la instrucción `input()` que ya vimos anteriormente.

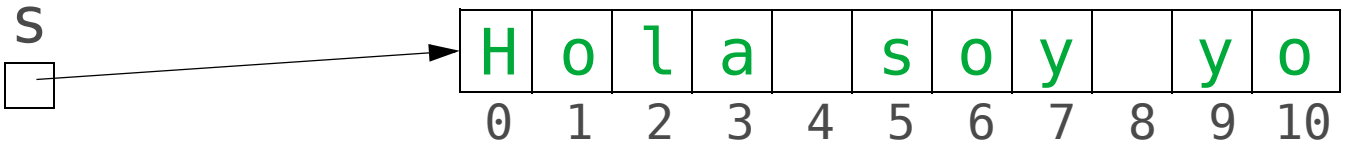
Como `input()` retorna un string, no es necesaria ninguna conversión de tipo.

Guardamos los valores retornados por las instrucciones `input()` en sendas variables, llamadas `nombre` y `padre`.

Finalmente, mostramos los textos leídos mediante una instrucción `print()` y un *f-string*.

Manipulación de strings

Los caracteres se numeran empezando en cero



Obtener el carácter número `i` de un string: `s[i]`

```
s[3] # "a"
```

Obtener un fragmento de un string: `s[i:j] # i incl., j excl.`

```
s[0:4] # "Hola"
```

Saber si un string contiene a otro:

```
'yo' in s # True
```

Obtener la longitud de un string:

```
len(s) # 11
```

Notas:

Podemos entender un texto como una secuencia de caracteres, de la que es posible obtener caracteres individuales o fragmentos.

Para identificar los caracteres, se usa un número llamado *índice*, que comienza por cero desde la izquierda, progresando hacia la derecha.

- En Python todos los índices se numeran desde *cero*.

Hay operaciones para:

- Obtener un carácter individual. Se obtiene como un string con un solo carácter.
- Obtener un fragmento, con los caracteres incluidos entre dos índices. Extrañamente, el primero de los índices está *incluido* en el fragmento, pero el segundo está *excluido*.
  - Si el segundo índice se omite, se entiende que es hasta el final del string. Por ejemplo, `s[9:]` vale `"yo"`.
  - Si el primer índice se omite, se entiende que es desde el principio del string. Por ejemplo, `s[:4]` vale "Hola".
- Saber si un texto está contenido en otro, obteniéndose un booleano.
- Obtener la longitud de un string, es decir su número de caracteres.
  - Si, por ejemplo un string tiene 14 caracteres, sus índices van de 0 a 13. El último carácter del string siempre tiene un índice que es uno menos que la longitud, debido al hecho de comenzar a contar desde cero.

Manipulación de strings (cont.)

Los strings tienen métodos, que son funciones que se aplican directamente al objeto con la notación

```
texto.método(parámetros)
```

Algunas operaciones usuales, siendo `s` un string

`s.find(sub)`	Retorna el primer índice en el que se encuentra `sub` dentro de `s`. Si no se encuentra, retorna `-1`
`s.strip()`	Retorna una copia de `s` sin los espacios en blanco iniciales ni finales
`s.lower()`	Retorna una copia en minúsculas de `s`
`s.upper()`	Retorna una copia en mayúsculas de `s`
`s.startswith(sub)`	Retorna `True` si `s` comienza por `sub`

Notas:

Los strings tienen muchas funciones o métodos asociados, que se invocan con la sintaxis:

```
string.función(parámetros)
```

Aquí se muestran algunas de estas funciones. La documentación de Python nos muestra aquí todas las funciones disponibles: https://docs.python.org/3/library/stdtypes.html#string-methods

Ejemplos, suponiendo que `s:str = "Esto es un string de prueba "`:

Operación	Resultado
`s.find("un")`	`8`
`s.find("una")`	`-1`
`s.strip()`	`"Esto es un string de prueba"`
`s.lower()`	`"esto es un string de prueba "`
`s.upper()`	`"ESTO ES UN STRING DE PRUEBA "`
`s.startswith("Este")`	`False`
`s.startswith("Esto")`	`True`

Recordemos que el string `s` nunca cambia. En las operaciones que retornan un string se obtiene uno nuevo.

Ejemplo de manipulación de strings

```
# -*- coding: utf-8 -*-
"""
Conversor de temperaturas entre Farenheit y Celsius

La conversión se hace con la fórmula: F=C*1.8+32

@author: Michael
@date:   Ene 2019
"""

def to_farenheit(cels: float)-> float:
    """
    Convierte Celsius a Farenheit

    Args:
        cels : Temperatura celsius
    Returns:
        El valor de cels en Farenheit
    """

    return cels*1.8+32
```

Ejemplo (cont.)

```
def to_celsius(faren: float)-> float:
    """
    Convierte Farenheit a Celsius

    Args:
        faren : Temperatura Farenheit
    Returns:
        El valor de faren en Celsius
    """

    return (faren-32)/1.8
```

Ejemplo (cont.)

```
def main():
    """
    Conversor entre temperaturas Farenheit y Celsius


    Se lee un texto que contiene la temperatura y la unidad,
    separados por un espacio en blanco. Ejemplos:
        20.0 C
        70.0 F
    A continuación se muestra el resultado en la otra unidad

    Por sencillez no hacemos detección de errores.
    Lo veremos más adelante
    """
```

Ejemplo (cont.)

```
# Leer la temperatura inicial
inicial: str = input("Temp. en C o F. Ejemplo:'30.0 C': ")
inicial = inicial.strip()

# Obtenemos la posición del espacio en blanco
indice_espacio: int = inicial.find(' ')

# Separamos el número y las unidades
temp: float = float(inicial[0:indice_espacio])
unidades: str = inicial[indice_espacio+1:].strip()

# Comparamos las unidades en mayúsculas
if unidades.upper() == 'C':
    # Celsius
    print(f"Temperatura: {to_farenheit(temp)} F")
else:
    # Entendemos que las unidades son Farenheit
    print(f"Temperatura: {to_celsius(temp)} C")
```

A observar en este ejemplo

Rodajas de string

- `s[i:j]`: incluye los caracteres de `i` a `j-1`
- `s[i:]`: incluye los caracteres desde `i` hasta el *último*

Uso de `strip()` para eliminar espacios extra al principio o al final

Comparación de strings en mayúsculas, para evitar diferencias entre minúsculas y mayúsculas

Uso de la instrucción condicional:

```
if expresion_booleana:
    instrucciones  # si True
else:
    instruciones   # Si False
```

Notas:

En este ejemplo hemos hecho un conversor de unidades de temperaturas de grados Celsius a Farenheit y viceversa.

El programa principal:

- Primero leerá del teclado un valor de temperatura y sus unidades con el formato consistente en un número, un espacio en blanco que sirve como separador, y una letra `C` o `F` que indica las unidades del valor inicial. Se lee todo como un *string*, mediante `input()`.
- A continuación, el programa separará las dos partes relevantes del dato leído: el valor numérico, que es un número real, y la letra que indica las unidades. Por sencillez, supondremos que el texto introducido es correcto.
  - Para realizar esta separación, buscaremos con `find()` el espacio en blanco, y obtendremos su índice. Ahora debemos obtener en primer lugar el fragmento inicial, con el número. Puesto que este fragmento es un texto, debemos convertirlo a número real con la conversión de tipo expresada como `float(valor)`.
  - En segundo lugar hay que obtener las unidades. Para ello tomamos el segundo fragmento del string original, desde la posición siguiente al indice que ocupa el primer espacio en blanco hasta el final del string. Observar que para llegar al final del string omitimos el segundo índice en el fragmento.
  - Por si acaso el usuario ha puesto más de un espacio en blanco de separación o ha puesto espacios en blanco después de la letra de unidad, aplicamos el método `strip()` para quitar esos espacios.

Notas:

- Una vez obtenida la letra con la unidad debemos tomar una decisión y hacer la conversión de Celsius a Farenheit, o al revés. Para tomar decisiones disponemos en Python de la instrucción **`if`**, con la sintaxis que vemos en el ejemplo, en la que cabe destacar:
  - tras la palabra **`if`** se pone una expresión que resulte en un valor booleano. Habitualmente una expresión relacional o lógica. En este caso, comparamos la variable `unidades` convertida a mayúsculas con la letra `'C'`. Se finaliza esta línea con `":"`.
  - Posteriormente se escriben las instrucciones a realizar en caso de que la expresión booleana sea cierta. En este caso, que las unidades sean Celsius. Podemos poner todas las instrucciones que queremos. Para saber cuándo acaban estas instrucciones usamos el sangrado o margen izquierdo. Al volver al nivel de sangrado original se entiende que esta parte de la instrucción **`if`** ha terminado.
  - A continuación se pone la palabra **`else`** (si no) seguida de `:`
  - Y por último se escriben las instrucciones a realizar en caso de que la expresión booleana sea falsa. De nuevo, usamos el sangrado para delimitar el ámbito de la parte **`else`**.
- En la instrucción **`if`** invocamos la función `to_farenheit()` para el caso de unidades Celsius o alternativamente la función `to_celsius()` para caso Farenheit. Mostramos los resultados en pantalla con un pequeño texto explicativo y sin olvidar las unidades del resultado.

Las funciones `to_farenheit()` o `to_celsius()` reciben como parámetro la temperatura y retornan el resultado de aplicar la fórmula de conversión de unidades.

Conversión de tipos

Compatibilidad de tipos: Python es un lenguaje con tipificación estricta

- No se pueden mezclar datos de distinta naturaleza en las expresiones
  - por ejemplo, números con strings
- Los números son compatibles entre sí

En ocasiones nos interesa cambiar el tipo de un dato

Conversión explícita de tipo:

```
tipo(dato)
```

Ejemplo:

```
numero = int(texto)
```

Hay que usarlas con precaución

Notas:

Hemos venido usando las conversiones de tipos en algunos de los ejemplos anteriores, por ejemplo:

- para convertir texto a número entero: `int(texto)`
- para convertir texto a número real: `float(texto)`
- para convertir un número a texto: `str(número)`

Aunque los números son compatibles entre sí, a veces interesa hacer alguna conversión explícita entre números

- Por ejemplo, de número real a entero: `int(número real)`.
- Esta conversión trunca la parte fraccionaria. No redondea. Por ejemplo, `int(6.8)` da `6`.
- Para redondear, existe la función `round()`. Por ejemplo, `round(6.8)` da `7`.

2.6. Uso de funciones matemáticas

El módulo `math` contiene constantes y funciones para hacer operaciones aritméticas con números reales (float)

- Existe un módulo similar llamado `cmath` para números complejos

Para usarlo debemos poner una instrucción `import` tras el *docstring* del módulo:

```
import math
```

Constantes:

`math.pi`	El número π
`math.e`	El número e
`math.inf`	+infinito
`math.nan`	*Not a number*, se usa para indicar errores

Notas:

Siempre que necesitemos usar las constantes π o e, debemos acudir a estas constantes del módulo `math`, pues están definidas con el máximo de decimales. Usar un valor como 3.1416 para el número π implica una grave pérdida de precisión. Ejemplo de uso:

```
# -*- coding: utf-8 -*-
"""
Contiene una función que muestra las constantes pi y e en pantalla

@author: Michael
@date    Feb 2021
"""

import math

def muestra_constantes():
    """
    Muestra las constantes pi y e en pantalla
    """
    print(f"{math.pi=}, {math.e=}")
```

En este ejemplo mostramos una facilidad de los *f-strings* disponible a partir de Python 3.8, que es la capacidad de mostrar un valor y un texto explicativo simplemente poniendo el símbolo = tras el valor.

- Resultado: `math.pi=3.141592653589793, math.e=2.718281828459045`

Operaciones matemáticas frecuentes

`math.sin(a), math.cos(a), math.tan(a)`	funciones trigonométricas (radianes)
`math.asin(v), math.acos(v), math.atan(v), math.atan2(y,x)`	trigonométricas inversas (radianes) de $-\pi/2$ a $\pi/2$ arco tangente de y/x entre $-\pi$ y π
`math.degrees(a), math.radians(a)`	conversiones de ángulos
`math.sinh(a), math.cosh(a), math.tanh(a)`	funciones trigonométricas hiperbólicas
`math.asinh(a), math.acosh(a), math.atanh(a)`	funciones trigonométricas hiperbólicas inversas
`math.exp(x), math.log(x), math.log10(x)`	e^x, logaritmo neperiano y logaritmo decimal
`math.sqrt(x)`	raíz cuadrada
`math.ceil(x), math.floor(x)`	redondeo por arriba y por abajo

Notas:

El módulo `math` permite operar principalmente con números reales. Algunas funciones, como `factorial()`, operan con enteros. No se admiten números complejos. Para estos números existe un módulo similar, llamado `cmath`.

Todas las funciones trigonométricas funcionan con *radianes*. Para operar con *grados* sexagesimales existen las funciones de conversión `degrees()` y `radians()`, que convierten respectivamente radianes a grados y grados a radianes. Ejemplos:

Expresión	Resultado
`math.sin(math.radians(30.0))`	`0.49999999999999994`
`math.degrees(math.asin(0.5))`	`30.000000000000004`

Puede observarse que la función arco tangente tiene dos versiones: `atan()` y `atan2()`.

- `atan()` opera solo en dos cuadrantes de la circunferencia, para ángulos entre $-\pi/2$ y $\pi/2$.
- `atan2()` es más precisa, pues opera en los cuatro cuadrantes. Utiliza los signos de `y` y `x` para determinar en qué cuadrante dar el resultado, retornando el ángulo que forma con el eje X el vector que va del origen al punto `(x, y)`. Así, `math.atan2(math.sin(x), math.cos(x))` siempre nos da `x`, si está entre $-\pi$ y π.

Operaciones matemáticas (cont.)

`math.factorial(x)`	factorial de x (debe ser entero no negativo)
`math.copysign(x,y)`	retorna x con el signo de y
`math.dist((x1, y1), (x2, y2))`	retorna la distancia entre dos puntos[ab] $\sqrt{(x2-x1)^2+(y2-y1)^2}$
`math.isinf(x)`	retorna True si x es ±infinito
`math.isnan(x)`	retorna True si x es nan
`math.isclose(a, b)`	retorna True si a y b están próximos; la tolerancia relativa por defecto es 1e-9, pero se puede cambiar con parámetros adicionales

a. Disponible a partir de Python 3.8
b. Se muestra un ejemplo con puntos de dimensión 2, pero la función permite también puntos de más dimensiones. Retorna la distancia euclidiana.

Notas:

Los números reales admiten valores especiales, como más o menos *infinito* y *nan* (*not a number*). Este último se usa para representar errores.

Estos valores tienen aritmética definida. Por ejemplo *nan* operando con cualquier cosa da como resultado *nan*. Infinito tiene la aritmética esperada. Por ejemplo, infinito más un número da infinito. Infinito menos infinito da *nan*.

Lo que estos valores no tienen bien definidos son los operadores de igualdad ==. El motivo es que pueden tener diferentes representaciones binarias, y la comparación de dos valores infinitos puede dar como resultado que son distintos. Por ello, para hacer la comparación con infinito o *nan* deben usarse las funciones `isinf()` e `isnan()`.

La comparación de números reales para su igualdad no es buena idea, ya que habitualmente pueden presentar errores de redondeo y, dos valores prácticamente iguales, pueden dar distinto al compararlos con el operador ==. Por ello, la función `isclose()` es muy interesante para ver si dos valores reales son muy parecidos.

Funciones predefinidas

Existen funciones predefinidas (que no forman parte de `math`), pero son útiles

Ya vimos `abs()` y `round()`. Algunas más son:

`max(x,y, ...)` `min(x,y, ...)`	máximo o mínimo de dos o más valores: ambas para cualquier valor numérico
`max(iterable)` `min(iterable)`	máximo o mínimo de los elementos de una lista, tupla o secuencia iterable
`sum(iterable)`	suma de los elementos de una lista, tupla o secuencia iterable

Notas:

Ejemplos de uso de estas funciones predefinidas:

Expresión	Resultado
`abs(-12.0)`	`12.0`
`round(6.8)`	`7`
`max((1, 6, 14, 12, 8))`	`14`
`min(11.0, 6.4, 5.8, -12.3)`	`-12.3`
`sum((1, 2, 1, 3))`	`7`

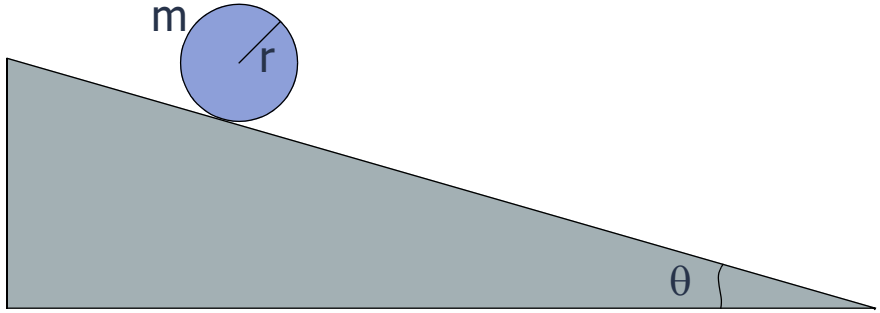
Para la función `max()` hemos usado un "*iterable*" que es esta tupla: `(1, 6, 14, 12, 8)`.

Similarmente, para la función `sum()` hemos usado la tupla: `(1, 2, 1, 3)`.

Para la función `min()`, en cambio, hemos puesto los parámetros separados por comas. También funcionaría con una tupla, añadiendo una pareja de paréntesis.

Ejemplo con funciones trigonométricas

Cálculo de movimientos de una esfera que rueda sobre un plano inclinado



Datos del ejemplo

I	momento de inercia	v	velocidad
m	masa	t	tiempo
r	radio	x	distancia
θ	ángulo del plano inclinado	E_{tras}	energía cinética de traslación
g	gravedad	E_{rot}	energía cinética de rotación
a	aceleración	ω	velocidad angular ($v=\omega r$)

Ecuaciones del movimiento

Las ecuaciones son (http://www.sc.ehu.es/sbweb/fisica3/):

$$I = \frac{2}{5}mr^2 \qquad a = \frac{g\sin\theta}{\left(1+\frac{I}{mr^2}\right)} \qquad v = at = \omega r$$

$$x = \frac{1}{2}at^2 \qquad E_{tras} = \frac{1}{2}mv^2 \qquad E_{rot} = \frac{1}{2}I\omega^2$$

Notas:

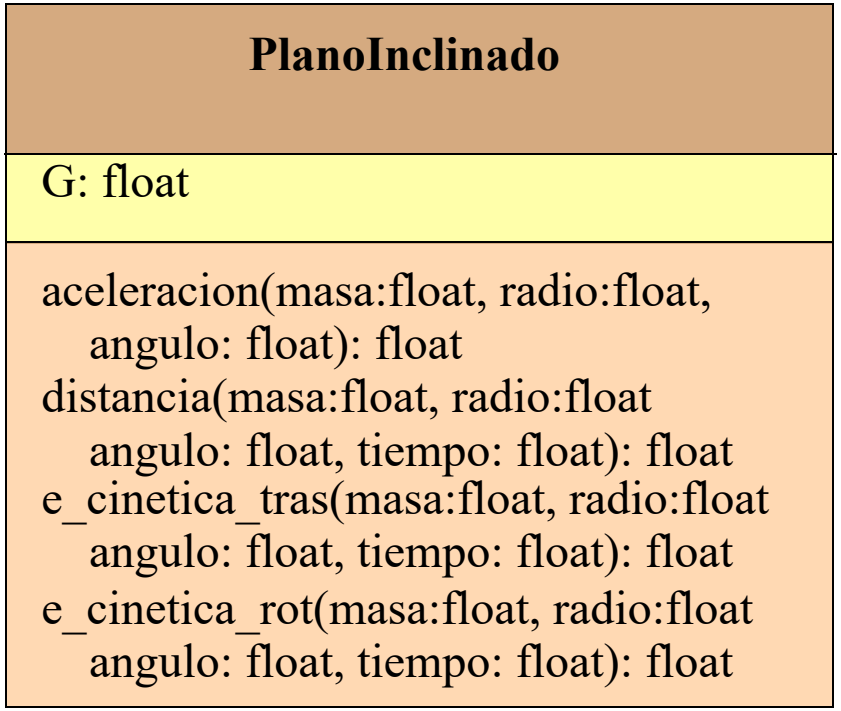
En este ejemplo queremos hacer un módulo con funciones que calculen algunas de las fórmulas que aparecen arriba.

Asimismo, crearemos un programa principal que muestre para una esfera de masa y radio fijos rodando en un plano inclinado de ángulo también fijo la distancia y energías a los cuatro segundos del inicio del movimiento.

Ejemplo: Diagrama de clases

Diseño del módulo

- la gravedad (G) es una constante
- funciones
  - cálculo de la aceleración
  - cálculo de la distancia
  - cálculo de la energía cinética de traslación
  - cálculo de la energía cinética de rotación



Ejemplo (cont.)

```
# -*- coding: utf-8 -*-
"""
Simula el movimiento de una esfera en un plano inclinado

Se entiende que la esfera rueda sin deslizarse
Utilizamos unidades del sistema internacional (Kg, m, s, J)
Para el ángulo utilizamos grados
Ecuaciones del movimiento en :
   http://www.sc.ehu.es/sbweb/fisica/

@author: Michael
"""

from typing import Final
import math

# Constantes
#   G : Gravedad en m/s**2

G: Final[float] = 9.8
```

Ejemplo (cont.)

```
def aceleracion(masa: float, radio: float,
                angulo: float)-> float:
    """
    Calcula la aceleración lineal del objeto (m/s)
    Args:
        masa: masa de la esfera, en Kg
        radio: radio de la esfera, en m
        angulo: angulo del plano inclinado, en grados
    Returns:
        la aceleración de la esfera
    """

    momento_inercia: float = 2.0*masa*radio**2/5.0
    return (G*math.sin(math.radians(angulo))/
            (1+momento_inercia/(masa*radio**2)))
```

Ejemplo (cont.)

```
def distancia(masa: float, radio: float, angulo: float,
              tiempo: float)-> float:
    """
    Calcula la distancia recorrida por el
    objeto en el tiempo indicado

    Args:
        masa: masa de la esfera, en Kg
        radio: radio de la esfera, en m
        angulo: angulo del plano inclinado, en grados
        tiempo transcurrido, en s
    Returns:
        la distancia recorrida por la esfera
    """

    return aceleracion(masa, radio, angulo)*tiempo**2/2.0
```

Ejemplo (cont.)

```
def e_cinetica_tras(masa: float, radio: float, angulo: float,
                    tiempo: float)-> float:
    """
    Calcula la energía cinética de traslación,
    del objeto transcurrido el tiempo indicado (Jul)

    Args:
        masa: masa de la esfera, en Kg
        radio: radio de la esfera, en m
        angulo: angulo del plano inclinado, en grados
        tiempo transcurrido, en s
    Returns:
        la energía cinética de traslación de la esfera en J
    """

    vel: float = aceleracion(masa, radio, angulo)*tiempo
    return masa*vel**2/2.0
```

Ejemplo (cont.)

```
def e_cinetica_rot(masa: float, radio: float, angulo: float,
                   tiempo: float)-> float:
    """
    Calcula la energía cinética de rotación,
    del objeto transcurrido el tiempo indicado (Jul)

    Args:
        masa: masa de la esfera, en Kg
        radio: radio de la esfera, en m
        angulo: angulo del plano inclinado, en grados
        tiempo transcurrido, en s
    Returns:
        la energía cinética de rotación de la esfera en J
    """

    # Observar que el cálculo del momento de inercia aparece
    # repetido. Deberíamos implementarlo con una función
    momento_inercia: float = 2.0*masa*radio**2/5.0
    vel_angular: float = (aceleracion(masa, radio, angulo)*
                          tiempo/radio)
    return momento_inercia*vel_angular**2/2.0
```

Ejemplo (cont.)

```
def main():
    """
    Programa principal que muestra para una esfera en
    un plano inclinado la distancia y energías a los
    cuatro segundos
    """

    # Crear el sistema plano-esfera
    # Ángulo 30 grados, esfera de 1.5 Kg y r=0.2 m
    masa: float = 1.5
    radio: float = 0.2
    angulo: float = 30

    # Mostrar resultados con 3 decimales
    # Ponemos ":.3f" dentro de las {} para indicar 3 decimales
    print("Dist. a los 4 seg: "+
          f"{distancia(masa, radio, angulo, 4.0):.3f} m")
    print("E. C. tras. a 4 seg: "+
          f"{e_cinetica_tras(masa, radio, angulo, 4.0):.3f} J")
    print("E. C. rot a 4 seg: "+
          f"{e_cinetica_rot(masa, radio, angulo, 4.0):.3f} J")
```

Notas:

En el ejemplo, las funciones simplemente realizan los cálculos necesarios y retornan el resultado. Necesitan como parámetros los datos del sistema (masa, radio y ángulo). Algunos de ellos también necesitan el tiempo.

El programa principal `main()` crea variables con los valores de los datos del sistema, y luego muestra en pantalla los resultados de invocar a tres de las funciones.

- Con objeto de no mostrar un alto número de decimales, utilizamos las facilidades de salida formateada de los *f-strings*. Dentro de una de las plantillas de datos puestas en el *f-string* entre llaves `{}`, añadimos la especificación de formato `:3f`, que indica que el número real (la `f` viene del tipo `float`) se debe presentar con tres decimales. Para otros números de decimales se puede cambiar el número 3. Por ejemplo, si quisiéramos mostrar el valor de la variable masa con 2 decimales, pondríamos una plantilla: `{masa:.2f}`

En general utilizamos unidades del sistema internacional, excepto para el ángulo, para el que nos resulta más natural usar grados.

- Lógicamente, en la función que usa el ángulo, que es `aceleracion()`, tenemos que convertirlo a radianes antes de aplicar la función trigonométrica seno: `math.sin(math.radians(angulo))`

Notas:

Tal como se indica en los comentarios del programa, en el diseño que se ha hecho se ha repetido el cálculo del momento de inercia, que hace falta en la función `e_cinetica_rot()` y en `aceleracion()`.

- Duplicar código no es una buena idea, pues hace que puedan crearse inconsistencias si se modifica el programa, por ejemplo porque en el futuro en lugar de usar una bola usemos un cilindro.
- Sería mejor definir una nueva función donde se realice el cálculo del momento de inercia una sola vez.
- Luego, usaríamos esa función invocándola tanto desde `e_cinetica_rot()` como desde `aceleracion()`.

La función del momento de inercia sería:

```
def momento_inercia(masa: float, radio: float)-> float:
    """
    Retorna el momento de inercia de una esfera de masa y radio
    indicados en los parámetros

    Args:
        masa (float): la masa de la esfera, en Kg
        radio (float): el radio de la esfera, en m.
    Returns:
        float: el momento de inercia, en kg*m**2.
    """
    return 2.0*masa*radio**2/5.0
```

Notas:

Y el uso desde una de las funciones, por ejemplo `aceleracion()`, sería:

```
    return (G*math.sin(math.radians(angulo))/
            (1+momento_inercia(masa, radio)/(masa*radio**2)))
```

Números aleatorios

El paquete `random` incluye operaciones para generar números aleatorios

Las más frecuentes:

`random.seed(), random.seed(a)`	inicializa el generador de números aleatorios con el reloj del sistema o con el número entero `a` ¡Hacer solo una vez!
`random.random()`	retorna un número aleatorio entre 0 y casi 1
`random.randint(start,stop)`	retorna un número aleatorio entero entre `start` y `stop` (incluidos)
`random.uniform(start,stop)`	retorna un número aleatorio real entre `start` y `stop` (incluidos)

Notas:

El uso de números aleatorios es muy frecuente en la programación.

- Muchos juegos deben tener situaciones en las que el programa haga una acción u otra de forma aleatoria.
- La simulación de procesos habitualmente requiere modelar eventos aperiódicos.
  - Suponer, por ejemplo, que pretendemos simular el funcionamiento de un conjunto de ascensores para evaluar la eficiencia energética de diversas estrategias de maniobra. Podemos simular la llegada de pasajeros que pulsan botones. Esta llegada no es siempre fija, sino que, como en la vida real, se produce con diferentes intervalos y probabilidades.

En la tabla anterior se muestran facilidades para generar números pseudoaleatorios.

- Un generador de números pseudoaleatorios se inicializa con un valor semilla y para una semilla concreta siempre proporciona la misma secuencia de números aparentemente aleatorios.
  - Si la semilla es variable, por ejemplo usando el reloj del sistema cuya hora siempre va cambiando, la secuencia obtenida será siempre diferente.
  - A veces nos interesa una secuencia fija, con objeto de poder repetir un experimento varias veces.
- La función `random.seed()` proporciona ambos comportamientos, según se llame con un parámetro o ninguno.
- Debe llamarse una sola vez, al principio del programa.
- Luego podremos obtener sucesivos números aleatorios con las funciones `random()`, `randint()` o `uniform()`.

Ejemplo con números aleatorios

En este ejemplo producimos una apuesta aleatoria para la bonoloto

- seis números del 1 al 49

```
# -*- coding: utf-8 -*-
"""
Producimos una apuesta aleatoria para la bonoloto

Son seis números aleatorios del 1 al 49
Para un funcionamiento correcto habría que eliminar repeticiones

@author: Michael
@date  : ene 2019
"""

import random
```

Ejemplo (cont.)

```
def main():
    """
    Muestra en pantalla seis números aleatorios entre 1 y 49
    """

    # Establecemos una semilla aleatoria
    random.seed()

    # Mostramos el número, una coma y sin salto de línea
    print(f"{random.randint(1, 49)}, ", end='')

    # Repetimos 4 veces más
    for _ in range(4):
        print(f"{random.randint(1, 49)}, ", end='')

    # La última vez no ponemos la coma
    print(random.randint(1, 49))
```

A observar en el ejemplo

Importancia de poner la semilla variable

- Si ponemos una fija, siempre saldrá la misma apuesta

Para hacer un `print` sin el salto de línea: `print(x, end="")`

Instrucción **`for`** para repetir unas instrucciones un número de veces determinado

- las instrucciones a repetir se especifican con el sangrado

El programa propuesto es limitado, ya que puede salir varias veces el mismo número, invalidando la apuesta

- Más adelante veremos cómo eliminar *repeticiones* de los números obtenidos, guardándolos en una lista o, mejor, en un conjunto

Notas:

En este ejemplo se ha ilustrado el uso de los números aleatorios. En primer lugar el programa ha establecido una semilla variable, mediante `random.seed()`. Posteriormente ha obtenido seis números enteros aleatorios comprendidos entre `1` y `49`, mediante la función `random.randint(1, 49)`.

Hemos visto la instrucción de bucle **`for`**, utilizada para repetir un conjunto de instrucciones un número determinado de veces: `4` en ese caso. Lo hacemos con la instrucción:

```
for _ in range(4):
```

La barra baja _ indica el nombre de una variable que sirve para contar las veces que vamos haciendo el bucle. Podemos poner cualquier nombre que queramos, pero si no lo vamos a usar en las instrucciones del bucle es mejor el nombre _, que implica que es un nombre sin importancia.

La construcción `range(4)` es un conjunto de `4` números enteros. La veremos más adelante, pero aquí nos basta saber que el número `4` es el número de repeticiones de las instrucciones del bucle.

Tras el símbolo `:` se ponen líneas con las instrucciones a repetir. Al igual que en otras instrucciones que ya hemos visto, el rango de la instrucción **`for`** se especifica con el sangrado, obligatorio. Cuando se vuelve al nivel de sangrado original, la instrucción **`for`** termina.

Como queremos poner todos los números aleatorios en la misma línea de pantalla y separados por comas, los cinco primeros llevan una coma detrás y el último se pone aparte, sin coma.

Además, como por defecto la instrucción `print()` añade un salto de línea al final y para los cinco primeros datos no queremos esto, añadimos al `print()` el parámetro `end=""`, que anula ese salto de línea.

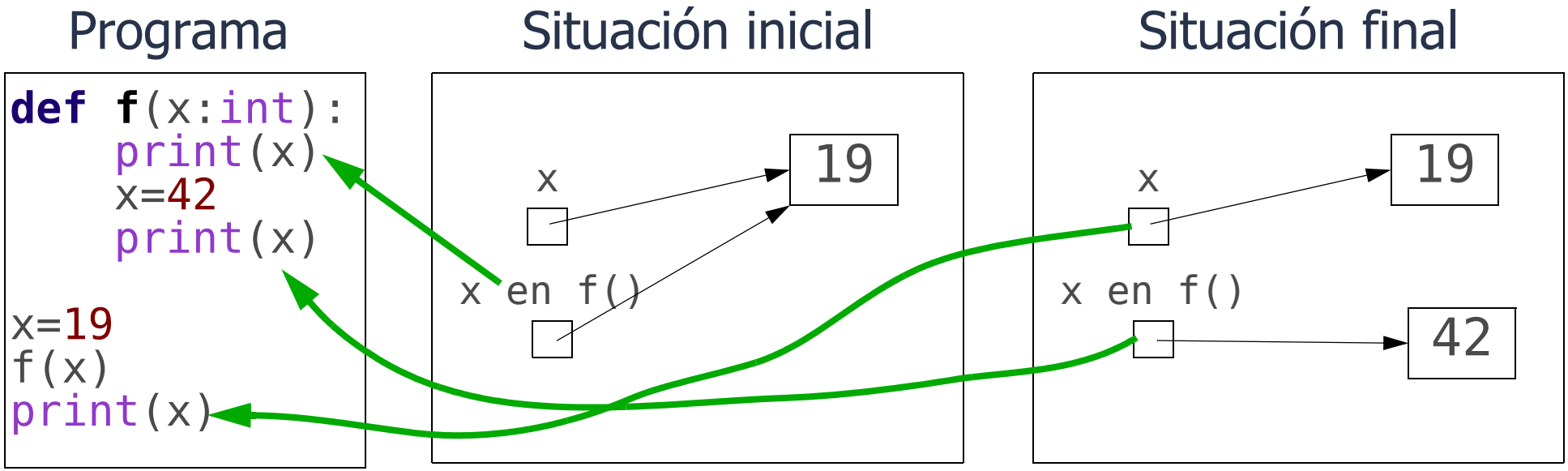
2.7. Variables y paso de parámetros

Como todas las variables, los parámetros de una función son referencias a objetos

- si el objeto original es *inmutable*, la función no lo puede cambiar
  - si se asigna otro objeto al parámetro, el original no cambia
  - por ejemplo números, booleanos, strings o tuplas
- pero si el objeto original es *mutable*, la función puede cambiarlo
  - el original cambia
  - por ejemplo, las listas

Ejemplo de paso de parámetros

Disponemos de este programa



La salida obtenida es

```
19
42
19
```

Notas:

En este ejemplo tenemos una función que recibe como parámetro un número entero, llamado x. Como toda variable, x es una referencia a un objeto. Esta variable existirá solo cuando se invoque a la función `f()`. No existirá antes de su ejecución y se destruirá a finalizar.

Por otro lado, el programa principal también tiene una variable entera llamada x. Hay que observar que esta variable, a pesar de su nombre, es diferente a la de la función. De hecho, podrían tener nombres diferentes.

La variable x del programa se crea mediante la asignación, al ejecutar `x=19`. En la figura intermedia podemos ver cómo apunta al objeto que contiene el dato 19.

Al invocar la función mediante `f(x)` se crea el parámetro x, indicado como "`x en f()`" en la figura intermedia. Como al invocar a la función se pasa entre paréntesis la variable x, el parámetro que se crea en ese momento apunta al mismo objeto que contiene el dato 19.

Cuando la función usa el nombre x se refiere siempre a su propio parámetro.

La función muestra el valor de x (que es 19), y luego cambia x para apuntar a un nuevo objeto que contiene el valor 42. Luego muestra ese valor en pantalla. Observar que este cambio afecta solo al parámetro, no a la variable x del programa. La situación es la de la figura de la derecha.

Por ello, cuando el programa muestra el valor de x está usando su propia variable (el parámetro ya dejó de existir al acabarse la función), y por ello muestra 19.

2.8 Listas y tuplas

La posibilidad de definir secuencias de objetos en la propia sintaxis de Python es una de sus características notables. Por ejemplo:

- tuplas

```
t=(1, 3, 7, "pepe")
```

- listas

```
l=[1, 3, 7, "pepe"]
```

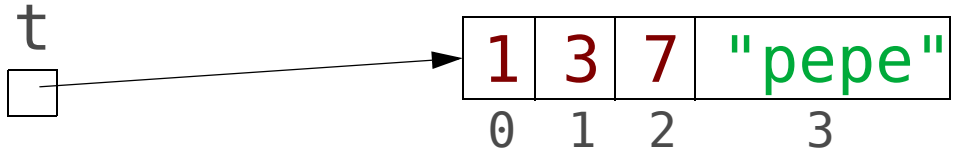
Tipos de secuencias

tipo	descripción	mutable
tuple	secuencia de elementos heterogéneos	no
list	secuencia de elementos heterogéneos	sí
str	secuencia de caracteres	no
range	secuencia de enteros seguidos	no

Numeración de los elementos

Las casillas de las secuencias se numeran comenzando por cero

- Llamamos *índice* a la numeración de las casillas



Puede haber listas y tuplas vacías

```
tupla_vacia=()
lista_vacia=[]
```

O de un solo elemento

```
t_uno = (3,) #observar la coma
l_uno = [4]
```

Operaciones comunes a las secuencias

Las listas y tuplas tienen las mismas operaciones que hemos visto para los strings; se muestran ejemplos con la tupla `t` y la lista `l` (pág. 112)

Operación	Descripción	Ejemplo	Resultado
`x in s`	`True` si la secuencia contiene a `x`	`"juan" in t`	`False`
`x + y`	Concatenar las dos secuencias	`(10,11)+(12,)`	`(10,11,12)`
`s * n`	Concatenar `s` a sí mismo `n` veces	`('a',)*3`	`('a','a','a')`
`s[i]`	Elemento `i`-ésimo de la secuencia	`l[1]`	`3`
`s[i:j]`	Rodaja de la secuencia, entre el índice `i` (incluido) y `j` (excluido)	`l[1:3]`	`[3, 7]`
`len(s)`	Número de elementos de `s`	`len(l)`	`4`
`min(s)`	Mínimo de los elementos de `s`	`min((1,5,3))`	`1`
`max(s)`	Máximo de los elementos de `s`	`max((1,5,3))`	`5`
`s.count(x)`	Número de ocurrencias de `x` en `s`	`l.count(2)`	`0`

Notación para obtener un elemento o un fragmento de una secuencia

En los fragmentos se puede *omitir* uno de los índices

Operación	Descripción	Ejemplo	Resultado
`s[i:]`	Rodaja de la secuencia, entre el índice `i` (incluido) y el final de la secuencia	`l[2:]`	`[7, "pepe"]`
`s[:j]`	Rodaja de la secuencia, entre el primer elemento y `j` (excluido)	`t[:3]`	`(1,3,7)`

Asimismo, para secuencias y fragmentos podemos usar *índices negativos*, que cuentan las casillas *desde el final*

Operación	Descripción	Ejemplo	Resultado
`s[-1]`	Último elemento	`l[-1]`	`"pepe"`
`s[-3:]`	Los tres últimos elementos	`t[-3:]`	`(3,7, "pepe")`

Operaciones con secuencias mutables

Las listas tienen además estas operaciones. Se ponen ejemplos con `l`

Operación	Descripción	Ejemplo	Resultado
`s[i]=x`	Reemplaza la casilla `i` de `s` por `x`	`l[1]='a'`	`[1,'a',7,'pepe']`
`s[i:j]=t`	Se reemplaza la rodaja `i:j` de `s` por los contenidos de `t`	`l[1:3]=(2,5)`	`[1,2,5,'pepe']`
`del(s[i:j])`	Se elimina la rodaja `i:j` de `s`	`del(l[1:3])`	`[1,'pepe']`
`s.append(x)`	Añade `x` al final de `s`	`l.append(8)`	`[1,3,7,'pepe',8]`
`s.clear()`	Borra todos los elementos de `s`	`l.clear()`	`[]`
`s.insert(i,x)`	Inserta `x` en `s`, en la casilla `i`	`l.insert(2,'b')`	`[1,3,'b',7, 'pepe']`

Notas:

Observar que al reemplazar un fragmento de una lista por otra secuencia:

- La secuencia no tiene por qué ser una lista. Puede ser una lista, una tupla, un rango, o un string.

Ejemplo	Resultado
`l[1:3]=(2,5)`	`[1,2,5,'pepe']`
`l[1:3]=[2,5]`	`[1,2,5,'pepe']`

- Los tamaños del fragmento y de la secuencia no tienen por qué coincidir. Si la secuencia es más grande, se abre hueco en la lista para los nuevos elementos.

Ejemplo	Resultado
`l[1:3]="hola"`	`[1, 'h', 'o', 'l', 'a', 'pepe']`

- Si es más pequeña, se reduce el tamaño de la lista.

Ejemplo	Resultado
`l[1:-1]=[]`	`[1, 'pepe']`

Operaciones con secuencias mutables (cont.)

Operación	Descripción	Ejemplo	Resultado
`s.remove(x)`	Elimina la primera aparición del elemento x	`l.remove("pepe")`	`[1,3,7]`
`s.reverse()`	Invierte los elementos de s	`l.reverse()`	`['pepe',7,3,1]`

Rangos

Son secuencias de números enteros

```
range(10) # 0,1,2,3,4,5,6,7,8,9
range(2,10) # 2,3,4,5,6,7,8,9
range(2,11,2) # 2,4,6,8,10, de dos en dos
```

Se usan para hacer bucles

```
for i in range(10): # i toma los valores de 0 a 9
    # las instrucciones se repiten 10 veces
    instrucciones
```

o crear listas o tuplas

```
l = list(range(10))
t = tuple(range(10))
```

Notas:

Observamos que en los rangos, si se pone segundo índice se entiende que el rango va desde el primer índice *incluido*, hasta el segundo, excluido.

Además, si se pone un *tercer índice* es el *paso* utilizado para ir aumentando los índices desde el primero.

Es muy habitual usar los índices para hacer bucles **`for`**, en los que las instrucciones del bucle se repiten un número conocido de veces.

- En estos bucles, la *variable de control* del bucle, `i` en el ejemplo de arriba, va tomando sucesivamente todos los valores del rango, y se puede usar en las instrucciones del bucle.

También se pueden usar los rangos para crear listas o tuplas. Usamos para ello la conversión de tipo:

- `list(secuencia)`, para convertir una secuencia a lista.
- `tuple(secuencia)`, para convertir una secuencia a tupla.

Ejemplo con lista, tupla y rango

```
# -*- coding: utf-8 -*-
"""
Trabaja con los nombres y días de los meses

@author: Michael
@date  : ene 2020
"""

# importamos estos módulos para poder hacer anotaciones de tipos
from typing import List
from typing import Tuple
```

Ejemplo con lista, tupla y rango (cont)

```
def main():
    """
    Muestra en pantalla los días de cada mes de 2021,
    con sus nombres
    """

    nombre_mes: List[str] = \
        ["enero", "febrero", "marzo", "abril", "mayo", "junio",
         "julio", "agosto", "septiembre", "octubre",
         "noviembre", "diciembre"]

    dias_mes: Tuple[int] = (31, 28, 31, 30, 31, 30, 31, 31,
                            30, 31, 30, 31)

    print("Días de cada mes de 2021:")
    for i in range(12):
        print(f"{nombre_mes[i]}: {dias_mes[i]}")
```

continuación de línea

Notas:

En este ejemplo hemos creado dos secuencias. Usamos una lista y una tupla a modo de ejemplo:

- una *lista* con los nombres de los 12 meses
- una *tupla* con los días de cada mes en un año no bisiesto.

Posteriormente creamos un bucle **`for`** para recorrer todos los índices de ambas secuencias, con un rango. Al poner `range(12)`, la variable `i` va tomando sucesivamente los valores `0, 1, 2, 3,`... hasta `11`.

- dentro del bucle mostramos con un `print()` el nombre del mes y el número de días de ese mes
- esto se repite para los 12 meses, y así obtenemos en pantalla:

```
Días de cada mes en 2021:
enero: 31
febrero: 28
marzo: 31
abril: 30
mayo: 31
junio: 30
julio: 31
agosto: 31
septiembre: 30
octubre: 31
noviembre: 30
diciembre: 31
```

A observar

Cuando una línea es muy larga se puede dividir en dos o más

- el carácter `\` sirve como continuación de línea
  - la línea posterior lleva un nivel de sangrado más, como en el ejemplo anterior
- tal como vimos en otros ejemplos, cuando queremos separar la línea dentro de `( )` o `[ ]` podemos hacerlo sin más, sin usar el carácter `\`
  - en ese caso, se sangra la nueva línea justo debajo del comienzo del paréntesis o corchete, como en el ejemplo anterior
- no podemos separar la línea en mitad de un nombre, o de un literal o de un string que no sea multilínea

Apéndice: Comprobación de anotaciones de tipo

Las anotaciones de tipo tienen como objetivo ayudar a la legibilidad y consistencia del código

- pero el intérprete Python no las comprueba

La herramienta `mypy` permite comprobar que el programa no viola los tipos anotados

Instalación desde el terminal `Anaconda prompt`:

```
conda install mypy
```

Uso desde el terminal `Anaconda prompt`:

```
mypy --ignore-missing-imports mi_fichero.py
```