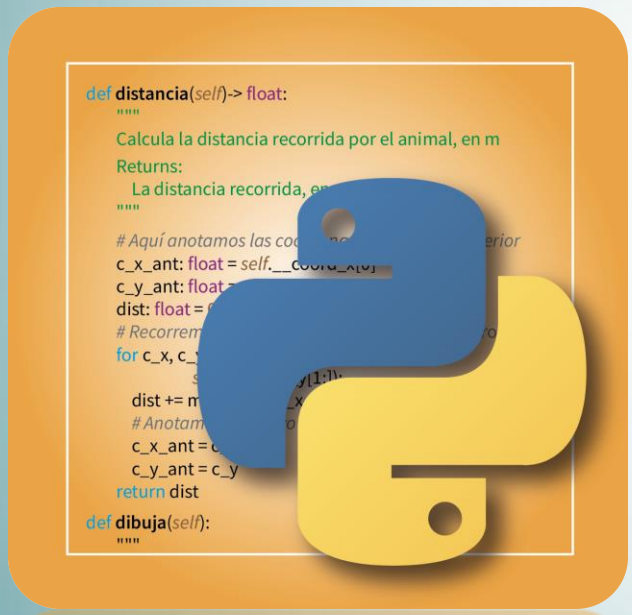
Programación

4. Estructuras algorítimas



Michael González Harbour

José Javier Gutiérrez García

José Carlos Palencia Gutiérrez

José Ignacio Espeso Martínez

Adolfo Garandal Martín

Departamento de Ingeniería Informática y Electrónica

Programación en Python

1. Introducción a los lenguajes de programación
2. Datos y expresiones
3. Clases
4. *Estructuras algorítmicas*

- Concepto de algoritmo. Instrucción condicional. Instrucción condicional múltiple. Instrucciones de bucle. Recursión. Descripción de algoritmos mediante pseudocódigo.

5. Estructuras de Datos
6. Tratamiento de errores
7. Entrada/salida
8. Herencia y polimorfismo

4.1. Concepto de algoritmo

Un algoritmo es:

- una secuencia finita de instrucciones,
- cada una de ellas con un claro significado,
- que puede ser realizada con un esfuerzo razonable
- y en un tiempo razonable

El algoritmo se diseña en la etapa de diseño detallado y se corresponde habitualmente con el nivel de operación, función o método

Los programas se componen habitualmente de muchas clases que contienen algoritmos, junto con datos utilizados por ellos

- los datos y algoritmos relacionados entre sí se encapsulan en la misma clase

Notas:

Un algoritmo corresponde al concepto de *secuencia de instrucciones*.

- En el mundo real, no informático, podríamos asimilarlo a una receta de cocina: una secuencia de pasos para elaborar un plato.
- Cada paso de la receta consiste en tomar uno o varios ingredientes y hacer algún proceso con ellos: pelar, triturar, calentar, etc.
- En el mundo informático se escribe una secuencia de instrucciones que el computador sabe hacer: sumar dos números, guardar un resultado en memoria, tomar una decisión en base a los datos, etc.

La segunda frase en el concepto de algoritmo es importante. Las instrucciones deben tener un *claro significado*, es decir, no ser ambiguas.

- En la vida corriente es habitual que las instrucciones que expresamos en lenguaje natural sean ambiguas o no completamente claras. A veces dependen del tono de voz o del contexto.
- Cuando describimos un algoritmo las instrucciones han de ser completamente precisas y deterministas.

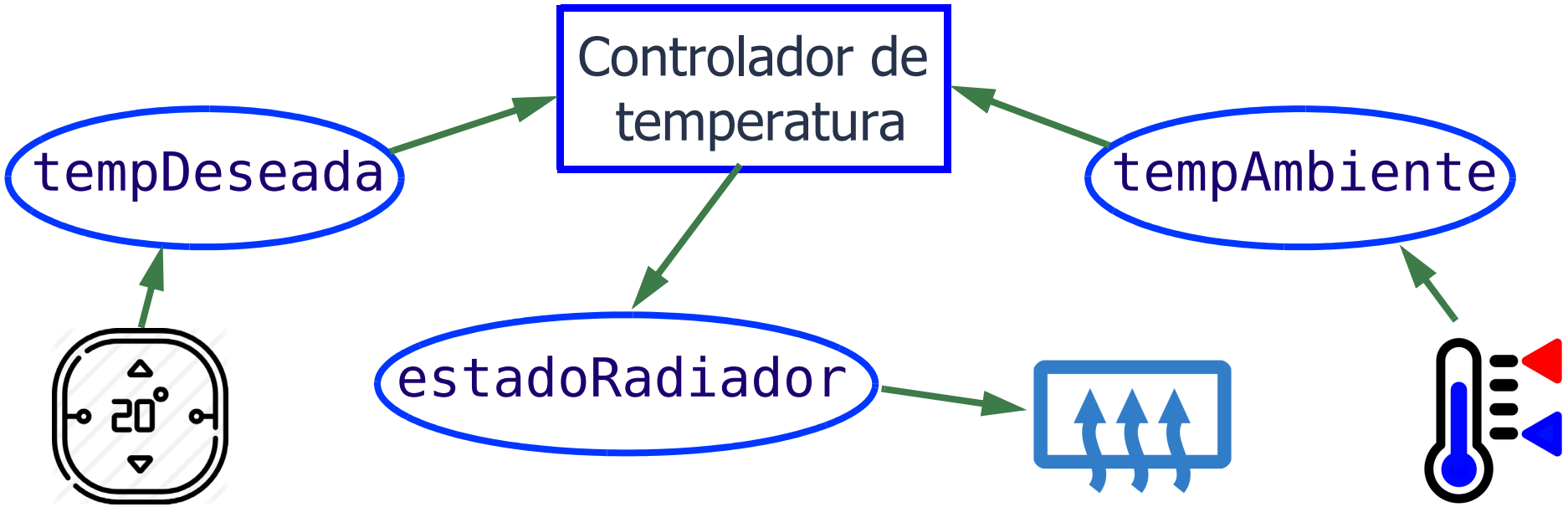
El esfuerzo y tiempo razonables son características muy importantes. Es muy fácil diseñar malos algoritmos que pueden tardar miles de años en completarse, incluso en los computadores más rápidos. Obviamente, los algoritmos así son inútiles.

- La definición de "razonable" es algo ambigua, pero debe interpretarse según el contexto. Por ejemplo, de nada nos sirve un algoritmo para predecir el tiempo que va a hacer mañana si tarda tres días en dar la solución.

Ejemplo de algoritmo

Algoritmo que intenta mantener una habitación a una temperatura deseada (± 0.5 grados)

- dispone de un radiador que se puede encender y apagar
- y un termómetro
- y una pantalla con botones para elegir la temperatura deseada



Notas:

Como ejemplo de un algoritmo mostramos las instrucciones de un sistema informático que trata de controlar un aparato de calefacción.

- Se intenta mantener la temperatura de la habitación a un valor especificado por el usuario, dentro de una tolerancia de 0.5 grados por arriba o por abajo.
- Esto se hace a base de encender o apagar un radiador eléctrico que calienta la habitación cuando está encendido.
- Se entiende que cuando el radiador está apagado la temperatura de la habitación disminuye, porque en el exterior hace frío.

La figura muestra los elementos del sistema y las tres variables o datos que utilizará nuestro algoritmo:

- `tempDeseada`: es una variable que contiene la temperatura deseada, y que ha sido seleccionada por el usuario en un termostato, que es un aparato externo a nuestro sistema.
- `tempAmbiente`: es una variable que contiene la temperatura real de la habitación, y que ha sido escrita por el termómetro contenido en el termostato.
- `estadoRadiador`: es una variable que indica si el radiador debe estar `encendido` o `apagado`. Esta variable deberá ser escrita por nuestro algoritmo.

En definitiva, el algoritmo leerá las variables `tempDeseada` y `tempAmbiente`, y escribirá valores `encendido`/`apagado` en la variable `estadoRadiador`.

Ejemplo (cont.)

Variables:

- `tempDeseada`: magnitud real (ºC)
- `tempAmbiente`: magnitud real (ºC)
- `estadoRadiador`: encendido o apagado

Estado del programa:

- descrito por los valores de `tempDeseada`, `tempAmbiente`, y `estadoRadiador` en cada instante

Ejemplo (cont.)

Algoritmo

```
repetir continuamente lo siguiente
   si hace frío encender el radiador
   si hace calor apagar el radiador
   esperar un rato
fin repetir
```

El algoritmo se repite continuamente (hasta que el usuario apague el sistema)

Ahora debemos refinar este algoritmo para expresarlo en términos de las variables del sistema

Notas:

Arriba se muestra una primera aproximación, de *alto nivel*, del algoritmo.

Esto aún no es un algoritmo real, ya que sus instrucciones no son lo suficientemente precisas

- ¿Qué significa "hace frío" o "hace calor"?
- ¿Cuánto es "un rato"?

Sin embargo esta primera aproximación al algoritmo es muy útil porque nos permite plasmar una estrategia que ahora refinaremos en un segundo paso, añadiendo el detalle necesario.

Observar que utilizamos una notación especial para indicar que unas instrucciones se deben repetir.

- Ponemos la palabra "`repetir`" y "`fin repetir`", y entre ellas las instrucciones que queremos repetir.
- Además, mediante el sangrado mostramos visualmente cuáles son las instrucciones a repetir.

Esta notación se denomina *pseudocódigo*, y la veremos más adelante. Pretende:

- Ser más sencilla que el código de un lenguaje de programación, al tener una gramática menos rígida, más libre.
- Ser independiente del lenguaje de programación en que se vaya a escribir el algoritmo.

Ejemplo (cont.)

Algoritmo refinado:

```
repetir continuamente lo siguiente
   si tempAmbiente<tempDeseada-0.5 entonces
      # hace frío
      estadoRadiador:=encendido
   fin si
   si tempAmbiente>tempDeseada+0.5 entonces
      # hace calor
      estadoRadiador:=apagado
   fin si
   esperar 1 minuto
fin repetir
```

Observar que si no hace ni frío ni calor el radiador se queda como estaba

Notas:

En el algoritmo que se muestra arriba se cumple la definición de algoritmo y todas las instrucciones son claras.

- Están descritas como usos y transformaciones de los tres datos que se manejan.

Ya hemos definido lo que significan conceptos como "hace frío", "hace calor", o "un rato".

A observar

Hemos descrito el algoritmo mediante la técnica llamada pseudocódigo, que tiene

- instrucciones de control presentes en todos los lenguajes

```
si condición entonces
   hacer cosas
fin si
```

  - obsérvese el uso del sangrado para determinar el ámbito de aplicación de cada instrucción de control
- cambios del valor de las variables, mediante cálculos si es preciso
- acciones expresadas sin el formalismo de los lenguajes
- comentarios, que en este caso empiezan por #

El propósito es que el pseudocódigo refinado sea sencillo, y directamente traducible a código en cualquier lenguaje

Estructuras algorítmicas

Las estructuras algorítmicas permiten componer instrucciones de un computador para que se ejecuten en el orden deseado

Estructura algorítmica	Descripción	Instrucción
Composición *secuencial*	Las instrucciones se ejecutan en secuencia, una tras otra	Ninguna. Simplemente se ponen las instrucciones una detrás de otra
Composición *alternativa*	En función de una condición se eligen unas instrucciones u otras	Instrucciones de control: *condicionales*
Estructura *iterativa*	Se repiten unas instrucciones mientras se cumple una condición	Instrucciones de control: *bucles*
Estructura *recursiva*	Se repiten unas instrucciones mediante una función que se invoca a sí misma	Función que se invoca a sí misma

Notas:

Sabemos que un algoritmo es una secuencia finita de instrucciones.

- En esta secuencia hay instrucciones que permiten definir en qué orden se ejecutan las demás instrucciones.

La gran capacidad de adaptación del computador para resolver una multitud de problemas se basa en su capacidad de tomar decisiones.

- Con las *instrucciones condicionales* se ejecutan unas instrucciones u otras en función de una o varias condiciones.

La gran potencia de cálculo del computador permite repetir cálculos muchas veces, a gran velocidad, a veces trabajando con gran cantidad de datos.

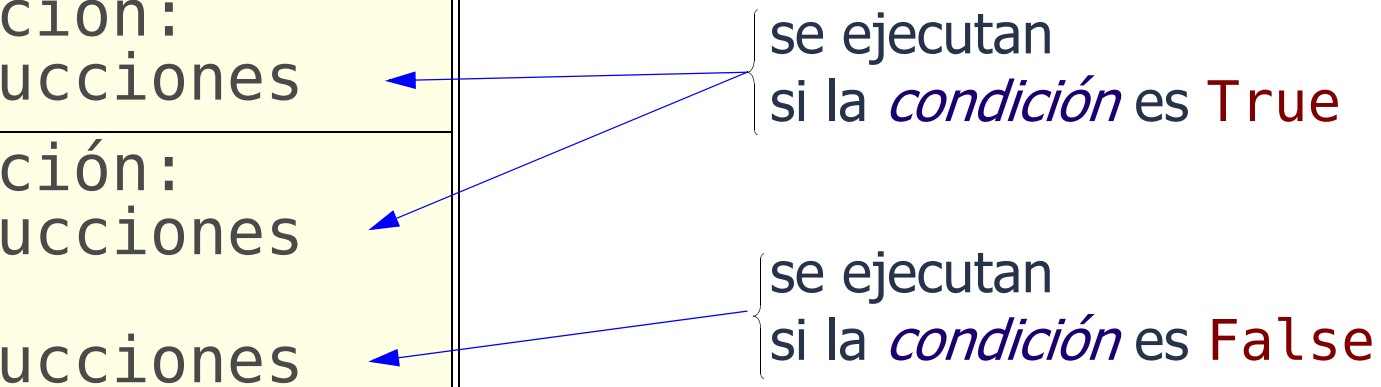
- Con las *instrucciones de bucle* se pueden repetir instrucciones tantas veces como se quiera, o mientras se cumpla una condición. A esto lo llamamos *iteración*.
- También se pueden repetir instrucciones encerrándolas en una función y haciendo que esa función se invoque a sí misma. A esto lo llamamos *recursión*.

4.2. Instrucción condicional simple

La instrucción condicional simple permite tomar decisiones empleando una variable booleana. Tiene dos modalidades:

modo simple	`if condición:` `  instrucciones`

modo alternativo	`if condición:`
	`  instrucciones`
	`else:`
	`  instrucciones`



La condición: expresión booleana (lógica o relacional)

El ámbito del contenido del **`if`** se indica con el sangrado

Notas:

Hay dos formatos para la instrucción `if`:

- Uno para ejecutar un conjunto de instrucciones si se cumple una condición, pero sin hacer nada si no se cumple:

```
if condición:
    instrucciones
```

- Otro con dos alternativas, una para el caso de que la condición sea `True`, y otra (la alternativa `else`) para el caso de condición `False`:

```
if condición:
    instrucciones
else:
    instrucciones
```

Observar la gramática de la instrucción `if`.

- Se pone el símbolo `:` después de la condición y después del `else`.
- El ámbito de las instrucciones que se ejecutan condicionalmente se expresa mediante el sangrado.
  - Es decir, cada alternativa termina al encontrar instrucciones sangradas al mismo nivel que el `if` o el `else`.

Ejemplo

Poner un texto aprobado o suspenso según la nota

```
if nota >= 5.0:
    print("Aprobado")
else:
    print("Suspenso")
```

Instrucciones condicionales anidadas

Las instrucciones `if` también se pueden anidar:

- con el sangrado siempre sabemos
  - el ámbito del contenido del **`if`**
  - a quién pertenece el **`else`**

Ejemplo: poner "*cum laude*" en el ejemplo anterior si `nota>=9`

```
if nota >= 5.0:
    print("Aprobado", end="")
    if nota >= 9.0:
        print(" cum laude")
    else:
        print()
else:
    print("Suspenso")
```

No se produce salto de línea

Añadimos el salto de línea que faltaba

Notas:

En el ejemplo hay dos instrucciones `if` anidadas, para tratar tres posibles condiciones para los valores de la variable `nota`:

`nota`	Resultado deseado
`0≤nota<5`	`"suspenso"`
`5≤nota<9`	`"aprobado"`
`9≤nota`	`"aprobado cum laude"`

Como las dos últimas alternativas comienzan con la palabra `"aprobado"`, ambas resuelven esta primera parte mediante una única condición (`nota>=5.0`).

- Tras la palabra `"aprobado"` no se pone un salto de línea, por si acaso luego hay que poner `" cum laude"`.

  - Para que el `print()` no añada un salto de línea se le pone el parámetro `end=""`

- Posteriormente, con la condición `nota>=9.0` se distingue entre las dos últimas situaciones y se pone el salto de línea que faltaba, o la palabra `" cum laude"` seguida de un salto de línea.

  - El salto de línea "a secas" se consigue con la instrucción `print()` sin parámetros.

Ejemplo: año bisiesto

```
# Leer el año del teclado
anyo: int = int(input("Año: "))
es_bisiesto: bool

# Determinar si es bisiesto
if anyo%4 == 0:
    if anyo%100 == 0:
        es_bisiesto = anyo%400 == 0
    else:
        es_bisiesto = True
else:
    es_bisiesto = False
```

Son bisiestos los años múltiplos de 4, excepto los múltiplos de 100 que no sean múltiplos de 400

x%n == 0
nos dice si x es divisible entre n

Notas:

En este ejemplo leemos del teclado un número entero que representa un año. Luego queremos calcular la variable `es_bisiesto` que determina si el año es bisiesto o no.

Hay varios casos, según el año sea múltiplo de 4, de 100 o de 400.

<table>
<tr><th>Múltiplo de 4</th><th>Múltiplo de 100</th><th>Múltiplo de 400</th><th>Es bisiesto</th></tr>
<tr><td rowspan="3">Sí</td><td rowspan="2">Sí</td><td>Sí</td><td>Sí</td></tr>
<tr><td>No</td><td>No</td></tr>
<tr><td>No</td><td></td><td>Sí</td></tr>
<tr><td>No</td><td></td><td></td><td>No</td></tr>
</table>

En los dos primeros casos (marcados en verde), una vez determinado que el año es múltiplo de 4 y de 100, el año es bisiesto si y solo si también es múltiplo de 400.

- Esto se determina con la instrucción relacional:

```
es_bisiesto = anyo%400 == 0
```

- Observar que esta condición cae dentro del caso "Cuándo no usar una instrucción `if`", en la pág. 23.

Ejemplo: año bisiesto (cont.)

Otras alternativas usando expresiones lógicas:

```
es_bisiesto: bool = (anyo%4 == 0 and anyo%100 != 0) or \
    anyo%400 == 0
```

o

```
es_bisiesto: bool = anyo%4 == 0 and \
    not(anyo%100 == 0 and not anyo%400 == 0)
```

Mostrar en pantalla el resultado usando *f-strings*:

```
if es_bisiesto:
    print(f"El año {anyo} es bisiesto")
else:
    print(f"El año {anyo} no es bisiesto")
```

Notas:

Es posible implementar el ejemplo del año bisiesto con operaciones lógicas, sin usar instrucciones `if`.

- Es más compacto, pero quizás más difícil de entender.
- No es necesariamente más eficiente, ya que la cantidad de comprobaciones a realizar es la misma.

Se muestra también cómo presentar el resultado del cálculo del año bisiesto en pantalla, usando mensajes diferenciados para los casos bisiesto y no bisiesto.

Cuándo no usar una instrucción `if`

A veces, al calcular expresiones booleanas la instrucción `if` es redundante. Ejemplos:

Caso redundante	Uso más eficiente

`if expr_booleana:`
`    resultado = True`
`else:`
`    resultado = False`
	`resultado = expr_booleana`

`if expr_booleana:`
`    return True`
`else:`
`    return False`
	`return expr_booleana`

Notas:

En ocasiones encontramos instrucciones `if` que son redundantes, pues se pueden sustituir por una expresión lógica o relacional sencilla.

- En estos casos es más eficiente no usar la instrucción `if`.
- El analizador de estilo de `Spyder` es capaz de identificar muchas de estas situaciones y avisarnos.

4.3. Instrucción condicional múltiple

Permite tomar una decisión de múltiples posibilidades, en función de un valor no booleano

- Algunos lenguajes tienen una instrucción (`switch` o `case`) muy eficiente que permite saltar directamente al caso deseado
- Pero Python no

Usaremos una "***escalera***" de instrucciones `if`

- la construcción `elif` encadena un `else` con el siguiente `if` y nos permite tener el sangrado uniforme
- un `else` final recoge los casos no tratados anteriormente

Instrucción condicional múltiple:

```
if condición1:
   instrucciones
elif condición2:
   instrucciones
elif condición3:
   instrucciones;
...
else:
   instrucciones
```

- Las condiciones se examinan empezando por la de arriba
- Tan pronto como una se cumple, sus instrucciones se ejecutan y la instrucción condicional finaliza
- Si ninguna de las condiciones es cierta se ejecuta la última parte `else`

Notas:

La instrucción `if` anidada permite hacer una "escalera" de condiciones para tomar una de entre múltiples alternativas.

La notación `elif` permite agrupar un `else` y el siguiente `if`, y hacer un *sangrado homogéneo* para todas las alternativas.

Instrucciones `if` convencionales

```
if condicion_1:
    alternativa_1
else:
    if condicion_2:
        alternativa_2
    else:
        if condicion_3:
            alternativa_3
        else:
            if condicion_4:
                alternativa_4
            else:
                alternativa_final
```

Instrucción equivalente, usando `elif`

```
if condicion_1:
    alternativa_1
elif condicion_2:
    alternativa_2
elif condicion_3:
    alternativa_3
elif condicion_4:
    alternativa_4
else:
    alternativa_final
```

Ejemplo: nota media con letra

```
class NotaAlumno:
    """
    Contiene la nota de un alumno y operaciones de conversión

    Attributes:
        __nota: la nota de un alumno, entre 0.0 y 10.0
    """

    def __init__(self, nota: float):
        """
        Constructor que pone la nota inicial

        Args:
            nota: la nota de un alumno, entre 0.0 y 10.0
        """

        self.__nota: float = nota
```

Ejemplo (cont.)

```
    def nota_letra(self)->str:
        """
        convierte la nota del alumno a una nota con letra
        y la retorna
        """

        nota_letra: str

        if self.__nota < 0.0 or self.__nota > 10.0:
            nota_letra = "Error"
        elif self.__nota < 5.0:
            nota_letra = "Suspenso"
        elif self.__nota < 7.0:
            nota_letra = "Aprobado"
        elif self.__nota < 9.0:
            nota_letra = "Notable"
        else:
            nota_letra = "Sobresaliente"

        return nota_letra
```

Notas:

En este ejemplo, la clase `NotaAlumno` contiene un único atributo que contiene la nota de un alumno, entre `0` y `10`.

La clase dispone del método `nota_letra` que retorna la nota de forma alfanumérica, usando la escala suspenso, aprobado, notable y sobresaliente.

En la instrucción `if` múltiple se usa una escalera de condiciones. La primera condición incluye los casos de error, en los que el valor de la nota es incorrecto.

Luego, las condiciones se evalúan en orden, comenzando por las notas más bajas hasta las más altas.

- Esto evita duplicar condiciones. Observar que, por ejemplo, la tercera condición, la de aprobado, es `nota<7.0`, no `5.0≤nota<7.0`. La condición `5.0≤nota` ya está garantizada por el `else` y las condiciones anteriores.
- Duplicar condiciones es ineficiente.

Ejecución automática del main()

Es cómodo ejecutar el `main()` de forma automática al cargar el módulo con ▶, sin tener que hacerlo manualmente

Para ello podemos incluir esta instrucción suelta después del `main()`:

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

Esto permite ejecutar el módulo directamente desde un *script* del sistema operativo o al cargarlo con ▶,

- pero evitando que el `main()` se invoque cada vez que alguien importe su módulo

La instrucción se basa en la existencia de la variable `__name__` cuyo valor es `"__main__"` si el módulo se ha ejecutado desde un *script* o al cargar el módulo con ▶

Notas:

A partir de ahora incluiremos al final de un módulo que tenga un `main()` la siguiente instrucción:

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

Esta instrucción permite proteger la ejecución del programa para que no se realice en una situación en la que no se debe.

4.4. Instrucciones de lazo o bucle

Permiten ejecutar múltiples veces unas instrucciones

- se corresponden a la ***composición iterativa***

La cantidad de veces se puede establecer mediante:

- ***una condición***:
  - se comprueba ***al principio***: las instrucciones del bucle se hacen cero o más veces
    - instrucción `while`
  - se comprueba ***al final***: las instrucciones del bucle se hacen una o más veces
    - hay lenguajes que tienen una instrucción para esto (`do-while`)
    - Python no, pero mostraremos cómo hacerlo
- ***un número fijo de veces***: se usa una variable de control
    - instrucción `for`

Notas:

En un bucle podemos repetir un conjunto de instrucciones tantas veces como queramos.

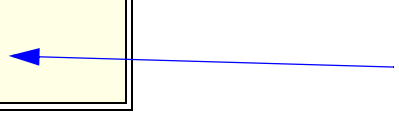
Nos encontramos diversas formas de establecer el número de veces:

- Un número fijo de veces: bucle con variable de control.
- Condición de permanencia, que se evalúa cada vez que repetimos el bucle; puede ser:
  - condición al principio: el bucle se hace cero o más veces,
  - condición al final: el bucle se hace una o más veces,
  - condición en mitad del bucle.

4.4.1. Bucle con condición de permanencia al principio

Es el bucle `while`:

```
while condicion:
    instrucciones
```



Se ejecuta cero o más veces

La condición se evalúa al principio, y cada vez que se completan las instrucciones

Notas:

En el bucle "mientras" o `while`, se define la condición de permanencia, que es una expresión booleana.

Tras la expresión se pone el símbolo `":"`

El ámbito de las instrucciones a repetir se marca con el sangrado.

- El ámbito del bucle acaba cuando encontramos una instrucción con el mismo sangrado que el bucle `while`.

Hay que darse cuenta de que la condición del bucle no se evalúa continuamente.

- Solo se evalúa al comenzar el bucle y cada vez que terminan sus instrucciones.

Ejemplo

Calcular el primer entero positivo tal que la suma de él y los anteriores sea mayor que 100

```
def main():
    """
    Programa que muestra en pantalla el primer entero
    tal que la suma de él y los anteriores sea mayor que 100
    """

    # suma contiene la suma de i y los anteriores positivos
    suma: int = 0
    # i cuenta las iteraciones; comenzamos por i=0
    i: int = 0
    while suma <= 100:
        i += 1
        suma += i

    # Mostrar el resultado
    print(f"La suma de i=1..{i} es {suma}"
```

Notas:

En este programa debemos calcular un sumatorio de los sucesivos números enteros.

$$suma = \sum_{i=1}^{n} i \;\Big|\; suma > 100$$

Para calcular un sumatorio de una función `f(i)` siempre se crea una variable (`suma` en este caso) para recoger el resultado de la suma.

- Esta variable se inicializa a cero, elemento neutro de la suma.

También creamos la variable `i`, que va contando los términos a sumar.

- Empezamos con un valor inicial igual a cero.
- En cada repetición del bucle añadimos 1 a la variable `i` y luego le añadimos `f(i)` a la variable `suma`.

El algoritmo para calcular el sumatorio, en pseudocódigo (ver el pseudocódigo en la pág. 82), es:

```
suma: entero = 0
i: entero = 0
mientras condicion
    i = i+1
    suma = suma+f(i)
fin mientras
```

Finalizado el bucle, `i` contiene las iteraciones realizadas y `suma` contiene el resultado buscado.

Notas:

Por ejemplo, si quisiéramos calcular el sumatorio de los 20 primeros cuadrados de los números enteros.

$$suma = \sum_{i=1}^{20} f(i) = \sum_{i=1}^{20} i^2$$

El algoritmo sería:

```
suma: entero = 0
i: entero = 0
mientras i < 20
    i = i+1
    suma = suma+i*i
fin mientras
```

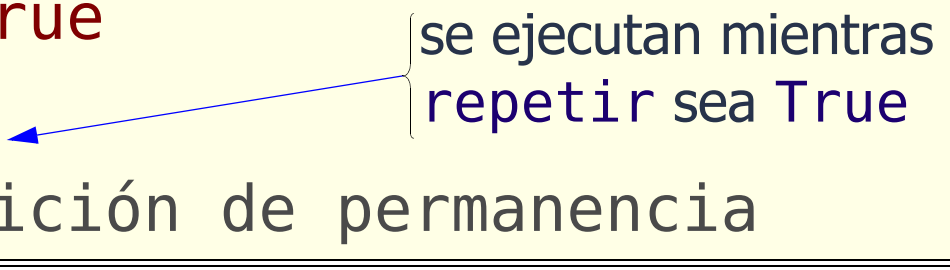
O equivalentemente podemos incrementar la `i` después de la suma:

```
suma: entero = 0
i: entero = 1
mientras i <= 20
    suma = suma+i*i
    i = i+1
fin mientras
```

4.4.2. Bucle con condición de permanencia al final

Aunque en Python no hay una instrucción para ello, se puede conseguir el efecto con esta construcción:

```
repetir: bool = True
while repetir:
    instrucciones
    repetir = condición de permanencia
```



Las instrucciones se ejecutan una o más veces

Notas:

El bucle con condición de permanencia al final es necesario cuando:

- Se desea garantizar que las instrucciones del bucle se hacen al menos una vez.
- O la condición de permanencia se calcula con datos que solo se conocen después de ejecutar las instrucciones del bucle.

Ejemplo

Calcular el máximo de unos números positivos introducidos por teclado, hasta que el usuario no quiera seguir

```
import math

def main():
    """
    Calcular el máximo de unos números

    Los números se introducen por teclado,
    hasta que el usuario no quiera seguir
    el resultado se muestra en pantalla
    """
```

Ejemplo (cont.)

```
# Contendrá el máximo encontrado hasta el momento
# Se inicializa a menos infinito (menor valor posible)
maximo: float = -math.inf # El mínimo valor posible

repetir: bool = True
while repetir:
    num: float = float(input("Número: "))
    if num > maximo:
        maximo = num
    print(f"El máximo hasta ahora es: {maximo}")
    seguir: str = input("Seguir? (s/n): ")
    repetir = seguir.lower() == "s"

print(f"El máximo es: {maximo}")
```

Notas:

En este ejemplo aplicamos un algoritmo de cálculo del máximo de una secuencia de números.

- En este caso los datos se introducen por teclado, pero el algoritmo de máximo se podría aplicar para otras secuencias de datos obtenidos de cualquier forma, por ejemplo en una lista o tupla.

En este algoritmo la variable `maximo` contiene el máximo valor encontrado hasta el momento.

- Esta variable se va comparando sucesivamente con todos los datos.
- Para que la primera comparación funcione, es preciso que esta variable valga inicialmente igual o menos que el mínimo valor posible:
  - Si los números fuesen positivos, este valor inicial podría ser cero.
  - Pero si son arbitrarios, debemos partir de menos infinito.

El pseudocódigo del algoritmo de máximo es:

```
maximo: real = menos infinito
repetir: booleano = True
mientras repetir
    num: real = obtener siguiente número de la secuencia
    si num > maximo entonces
        maximo = num
    fin si
    repetir = True si la secuencia no ha terminado, False si no
fin mientras
# maximo contiene el resultado final
```

Ejemplo: alternativa

Si *no* nos preocupa la memoria podemos meter los números en una lista y usar la función predefinida `max()`:

```
from typing import List
def main():
    """
    Calcular el máximo de unos números
    """

    # Lista que contendrá los números
    lista: List[float] = []
    repetir: bool = True
    while repetir:
        num: float = float(input("Número: "))
        lista.append(num)
        seguir: str = input("Seguir? (s/n): ")
        repetir = seguir.lower() == "s"

    print(f"El máximo es: {max(lista)}")
```

Notas:

En esta alternativa al algoritmo de máximo metemos todos los elementos en una lista y luego usamos la función predefinida max().

- Inicialmente creamos la lista vacía = []
- Usamos el método append de la lista para añadir elementos al final.

Esto solo funciona para listas de números, textos o datos con una relación de orden definida.

4.4.3. Bucle con salida en medio

En ocasiones la condición de permanencia (o de salida) está en mitad del bucle

En ese caso usamos un bucle *infinito* y ponemos la condición de salida con la instrucción `break`, que abandona el bucle

```
while True:
    instrucciones
    if condición de salida:
        break;
    otras instrucciones
```

abandonar el bucle

Notas:

Para hacer un bucle infinito o indefinido ponemos en el bucle "mientras" una condición que siempre vale `True`.

Usamos la instrucción `break` para abandonar el bucle.

Ejemplo: bucle con salida en medio

Usaremos las clases `Lectura` y `Escritura` del paquete `fundamentos`, para hacer la lectura de datos y mostrar resultados

```
import math
from fundamentos.lectura import Lectura
from fundamentos.escritura import Escritura

def main():
    """
    Calcula distancias entre puntos del globo terráqueo

    El cálculo se hace con la fórmula del círculo máximo
    El programa lee las distancias de una ventana, muestra
    resultados en otra, y repite el proceso continuamente
    """

    # Paso 1: crear objetos de las clases Lectura y Escritura
    lec = Lectura("Distancias entre ptos. del globo terráqueo")
    esc = Escritura("Resultado de distancia")
```

Ejemplo (cont.)

```
    # Paso 2: crear las entradas para leer datos
    lec.crea_entrada("Latitud 1 (º) : ", 0.0)
    lec.crea_entrada("Longitud 1 (º) : ", 0.0)
    lec.crea_entrada("Latitud 2 (º) : ", 0.0)
    lec.crea_entrada("Longitud 2 (º) : ", 0.0)

    while True:

        # Paso 3: esperar a que el usuario teclee
        finalizar: bool = lec.espera()
        if finalizar:
            break

        # Paso 4: leer los valores tecleados
        lat1: float = lec.lee_float("Latitud 1 (º) : ")
        lon1: float = lec.lee_float("Longitud 1 (º) : ")
        lat2: float = lec.lee_float("Latitud 2 (º) : ")
        lon2: float = lec.lee_float("Longitud 2 (º) : ")

        # Cálculo del resultado
        lat1 = math.radians(lat1)
```

Ejemplo (cont.)

```
        lat2 = math.radians(lat2)
        lon1 = math.radians(lon1)
        lon2 = math.radians(lon2)
        dist: float = math.degrees(math.acos(math.sin(lat1)* \
            math.sin(lat2)+math.cos(lat1)* \
            math.cos(lat2)*math.cos(lon1-lon2)))* \
            60.0*1.852
        esc.inserta_valor("La distancia (km)", dist)

    # Paso 5: Destruir las ventanas
    lec.destruye()
    esc.destruye()
```

Notas:

En este ejemplo escribimos un programa que es una calculadora de distancias en el globo terráqueo.

- El programa pregunta repetidas veces al usuario las latitudes y longitudes de dos puntos,
- y para cada pareja de puntos muestra el resultado de la distancia.

Usamos un objeto de la clase `Lectura` para leer datos de teclado usando una ventana.

Y usamos un objeto de la clase `Escritura` para mostrar resultados usando una ventana.

El uso de una interfaz gráfica con ventanas da un aspecto más amigable a la entrada/salida de datos.

Las clases `Lectura` y `Escritura` pertenecen al paquete `fundamentos`, que puede descargarse de moodle.

A continuación veremos el uso de estas clases.

Notas:

Clase `Lectura`. Estos son los pasos a realizar para usarla:

- Paso 1: crear un objeto de la clase `Lectura`:

```
lec = Lectura("Título de la ventana")
```

- Paso 2: crear una o varias "cajitas" o *entradas* de datos en la ventana:

```
lec.crea_entrada("Etiqueta explicativa", valor_inicial)
```

- Paso 3: esperar a que el usuario teclee:

```
finalizar: bool = lec.espera()
# la variable finalizar será True si se ha cerrado la ventana
```

- Paso 4: leer los datos tecleados; pueden ser reales, enteros o textos:

```
variable1: float = lec.lee_float("Etiqueta explicativa")
variable2: int = lec.lee_int("Etiqueta explicativa")
variable3: str = lec.lee_str("Etiqueta explicativa")
```

La "etiqueta explicativa" al leer la entrada debe coincidir exactamente con la usada para crearla.

Posteriormente podemos trabajar con las variables leídas.

En la calculadora de distancias los pasos 1 y 2 se hacen una sola vez y están fuera del bucle.

- En cambio los pasos 3 y 4 se repiten cada vez que leemos datos y por tanto están dentro del bucle.

Al acabar hay que dar un último paso, para destruir la ventana:

```
lec.destruye()
```

Notas:

Clase `Escritura`: Estos son los pasos a realizar para usarla:

- Paso 1: crear un objeto de la clase `Escritura`:

```
esc = Escritura("Título de la ventana")
```

- Paso 2: crear una o varias "cajitas" de datos en la ventana:

```
esc.inserta_valor("Etiqueta explicativa", valor)
```

- Paso 3: esperar a que el usuario haya visto los resultados:

```
finalizar: bool = esc.espera()
# la variable finalizar será True si se ha cerrado la ventana
```

Al acabar hay que dar un último paso, para destruir la ventana:

```
esc.destruye()
```

4.4.4 Bucle con variable de control

Es el bucle `for`:

```
for variable in secuencia:
    instrucciones
```

La secuencia puede ser un rango, tupla, lista, ...

Notas:

Cuando sabemos de antemano el número de veces que haremos un bucle usaremos un bucle con variable de control.

- La variable de control cuenta el número de veces que hemos hecho el bucle,
- o recorre uno por uno todos los valores de una secuencia que queremos visitar.

Cuando nos encontramos en estos casos no debemos usar un bucle `while`.

- Así será más fácil leer el programa,
- y más difícil equivocarse.

Ejemplos

Suma de los cuadrados de los 20 primeros enteros positivos:

```
suma: int = 0
for i in range(1, 21):
    suma += i**2
```

También para incrementos distintos de uno (ej: nº pares):

```
suma: int = 0
for i in range(2, 21, 2):
    suma += i**2
```

Los argumentos de la función `range` incluyen el comienzo y excluyen el final

```
range(5)          # 0, 1, 2, 3, 4
range(1, 5)       # 1, 2, 3, 4
range(1, 10, 2) # 1, 3, 5, 7, 9
```

Notas:

Los ejemplos muestran la forma preferida para calcular sumatorios si el número de términos a sumar es conocido.

El ejemplo de la pág. 39, con un sumatorio de los primeros 20 cuadrados de los números enteros, se haría mejor con un bucle `for`, en lugar de un bucle `while:`

```
suma: entero = 0
para i desde 1 hasta 20
    suma = suma+i*i
fin para
```

Comprensiones de listas (*list comprehensions*)

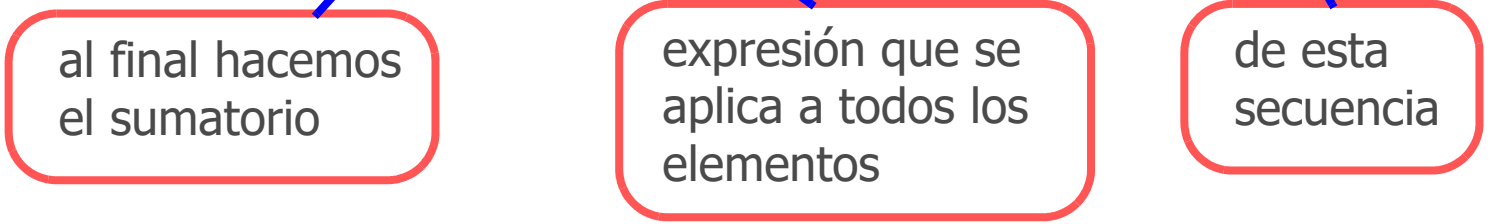
Los ejemplos anteriores se pueden expresar mediante "comprensiones de listas"

- son construcciones en las que se forma una lista aplicando una expresión a todos los elementos de una secuencia

```
suma: int = sum([i**2 for i in range(1, 21)])
```

list comprehension

```
suma: int = sum([i**2 for i in range(2, 21, 2)])
```



Notas:

Una comprensión de lista es un mecanismo para crear una lista nueva a partir de otra, aplicando una operación a cada uno de los elementos de la lista original.

Por ejemplo, supongamos una lista de palabras:

```
l: List[str] = ["uno", "dos", "tres"]
```

Ahora queremos obtener una nueva lista igual a la original pero con las palabras en mayúsculas. Aplicamos el método `upper()` a cada elemento de la lista original. La variable de control `pal` va recorriendo uno por uno esos elementos.

```
mayus: List[str] = [pal.upper() for pal in l]
```

La nueva lista valdrá:

```
['UNO', 'DOS', 'TRES']
```

En el ejemplo de la pág. 59, la nueva lista creada con una comprensión de lista contiene los cuadrados de los números enteros. Luego aplicamos a la nueva lista la operación `sum()`, que suma todos sus elementos.

Variantes de bucles

Hacia atrás:

```
for i in range(21, 1, -1):          # de 21 a 2
for i in reversed(range(1, 21)): # de 20 a 1
```

Vacío:

```
for j in range(0, finish):      # si finish<0
```

Anidado

```
for i in range(10):
    for j in range(20):
        ...
```

Notas:

El primer ejemplo muestra un bucle en el que se recorren los índices de un rango hacia atrás, disminuyendo de uno en uno.

- En la primera variante se pone un paso `-1` en el rango.
- En la segunda variante se pone un rango normal, con paso `1`, pero luego se le da la vuelta con la operación `reversed()`.
  - La operación `reversed()` funciona también con listas, tuplas y otras secuencias, pero no modifica la lista ni crea una nueva, sino que retorna un *iterador*, que es un objeto que la recorre al revés.
  - Un iterador es muy eficiente, pero solo se puede usar en un recorrido, como por ejemplo en un bucle `for`.
  - Por ejemplo, este bucle muestra las letras de `"pepe"` una por una, al revés:

```
for i in reversed("pepe"):
    print(i)
```

El segundo ejemplo muestra que el bucle for puede hacerse cero veces, si el rango que se le pasa no tiene ningún valor.

El tercer ejemplo muestra que pueden hacerse bucles anidados.

- En este caso las instrucciones del segundo bucle se realizan 10*20 = 200 veces.

Ejemplo: Gráficas de funciones reales

El módulo `matplotlib.pyplot` contiene facilidades para crear gráficos X-Y avanzados

- los puntos se pasan metidos en listas
- se pueden mostrar como puntos o líneas de diversos colores
- se pueden mostrar varios gráficos en la misma ventana

Para más información sobre formatos, colores, etc:

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

El módulo `numpy` contiene facilidades para trabajar con vectores, matrices y otras estructuras de datos

- usaremos su función `linspace()`, que permite obtener una secuencia de `n` números reales en un rango `[start, stop]`

```
numpy.linspace(start, stop, n)
```

Ejemplo (cont.)

```
# -*- coding: utf-8 -*-
"""
Programa que muestra gráficas de funciones trigonométricas

@author: Michael
@date  : feb 2019
"""

from typing import List
import math
import numpy as np
import matplotlib.pyplot as plt


def main():
    """
    Muestra gráficas del seno y coseno entre 0 y 3 PI
    """
```

Ejemplo (cont.)

```
# creamos tres listas: el eje X, el seno y el coseno
list_x: List[float] = np.linspace(0, 3*math.pi, 200)
list_seno: List[float] = []
list_coseno: List[float] = []
# añadimos a las listas del seno y coseno sus valores
for varx in list_x:
    list_seno.append(math.sin(varx))
    list_coseno.append(math.cos(varx))

#Creamos la gráfica para dibujar el seno
plt.plot(list_x, list_seno, 'red', label="seno")
# y para el coseno
plt.plot(list_x, list_coseno, 'blue', label="coseno")
# Decoraciones
plt.ylabel('f(x)')
plt.xlabel('x')
plt.title('Funciones trigonométricas')
plt.grid(True)
plt.legend()
# Mostrar las gráficas
plt.show()
```

Notas:

En este ejemplo creamos una gráfica mostrando dos funciones en la misma gráfica: las funciones seno y coseno de los ángulos entre 0 y 3π.

Para crear la gráfica necesitamos crear tres listas:

- Una llamada `list_x` con los ángulos entre 0 y 3π, con un total de 200 puntos, para que las curvas de la gráfica sean suaves.
  - La creamos con la función `linspace()`, del paquete `numpy`, que permite crear rangos parecidos a los de la operación `range()`, pero con números reales.
- Otra con el seno de esos ángulos.
- Otra con el coseno de los ángulos.
  - Estas dos últimas listas se crean inicialmente vacías y luego mediante un bucle recorremos la lista `list_x` y para cada ángulo metemos su seno y su coseno, respectivamente en su correspondiente lista; usamos para ello el método `append()` que añade un elemento al final de la lista.

Con las listas ya formadas creamos las dos gráficas usando la función `plot()`. Le pasamos la lista del eje x y la lista del eje y, así como el nombre del color en inglés y la etiqueta que identifica la gráfica.

Creadas las listas, establecemos algunas decoraciones, tales como las etiquetas de los ejes, el título de la gráfica, la rejilla de la gráfica, y la leyenda.

Finalmente mostramos la gráfica con la función `show()`.

4.4.5. Instrucciones de salto en bucles

Hay tres instrucciones para saltarse las instrucciones restantes del bucle

- `break`
  - se sale del bucle
- `continue`
  - se salta el resto de las instrucciones del bucle, pero sigue en él
- `return`
  - termina la función; si estamos en un bucle, lógicamente también se sale de él

Se recomienda no usarlas en bucles `while`; es más ortodoxo que las condiciones de permanencia se expresen en la propia instrucción `while`

- el `break` se pondría con condiciones adicionales de permanencia en el bucle
- el `continue` se resuelve con una instrucción condicional

Notas:

Se muestran tres instrucciones que permiten abandonar anticipadamente un bucle. Lógicamente solo tiene sentido usarlas dentro de una instrucción condicional `if`.

En general se recomienda no usarlas en bucles `while`, aunque podemos hacerlo en los siguientes casos:

- El `break` se puede usar para bucles con la salida en medio, tal como hemos visto en la pág. 47.
- El `return` se puede usar cuando la función no es muy larga y de un vistazo podemos entender bien el funcionamiento.

Se recomienda no usar el `continue`, porque es innecesario y es más ortodoxo usar un `if`, como se muestra.

```
while condicion:
    instrucciones
    if ir_al_final:
        continue
    otras_instrucciones
```

```
while condicion:
    instrucciones
    if not ir_al_final:
        otras_instrucciones
```

4.5. Recursión

Muchos algoritmos iterativos pueden resolverse también con un algoritmo ***recursivo***

- el algoritmo se invoca a sí mismo
- en ocasiones es más natural
- en otras ocasiones es más engorroso

El ***diseño recursivo*** consiste en diseñar el algoritmo mediante una estructura condicional de dos ramas

- ***caso directo***: resuelve los casos sencillos
- ***caso recursivo***: contiene alguna llamada al propio algoritmo que estamos diseñando

Notas:

Una función que se invoca a sí misma repite sus instrucciones. Este mecanismo se llama *recursión* y la función se dice que es *recursiva*.

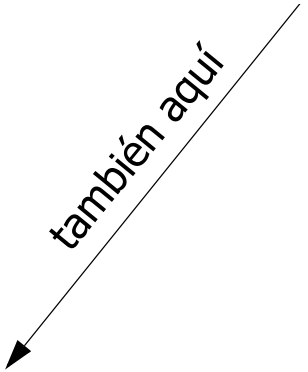
Para que las repeticiones sean finitas, es preciso poner la llamada a la propia función en una instrucción condicional.

Si la función no retorna ningún valor, un esquema habitual es:

```
def funcion_recursiva(parámetros):
    if caso_directo:
        instrucciones_caso_directo
    else:
        instrucciones_caso_recursivo
        funcion_recursiva(parametros)  # llamada a la misma función
        más_instrucciones_si_hacen_falta
```

Si la función retorna un valor, un esquema habitual es:

```
def funcion_recursiva(parámetros)->tipo_retornado:
    if caso_directo:
        instrucciones_caso_directo
        return valor_caso_directo
    else:
        instrucciones_caso_recursivo
        return operación_que_invoca_a funcion_recursiva(parametros)
```



- La función retorna lo que a su vez retorna la siguiente invocación a la propia función.

Ejemplo

Definición iterativa para el cálculo del sumatorio de una función de un número natural

$$sum(n) = \sum_{i=1}^{n} f(i)$$

Definición iterativa

$$sum(0) = 0$$

$$sum(n) = f(1) + f(2) + \ldots + f(n), \quad n \geq 1$$

Ejemplo

Definición recursiva

$$sum(0) = 0$$

$$sum(n) = f(n) + sum(n-1), \quad n \geq 1$$

La definición es correcta pues el número de recursiones es finito

Ejemplo: sumatorio recursivo

```
def sumatorio(n: int)-> float:
    """
    Calcula el sumatorio de f(i) desde i=1 hasta n

    Args:
        n: el número de términos a sumar

    Returns:
        el sumatorio de f(i) desde i=1 hasta i=n
    """

    if n == 0:
        # Caso directo: no hay nada que sumar
        return 0
    else:
        # Caso recursivo
        return f(n)+sumatorio(n-1)
```

Notas:

Podemos observar que en este ejemplo la función `sumatorio()` retorna `f(n)` sumado a lo que a su vez retorna la misma función `sumatorio()` pero aplicada a `n-1`.

Como la función se invoca cada vez con un parámetro que es un número cada vez menor, si hemos partido de un número positivo eventualmente invocaremos a la función para `n=0`.

- En esa llamada se cumple el caso directo, en el que el sumatorio es cero.

Esta tabla muestra los sucesivos valores de los parámetros y del valor retornado en cada invocación, si la primera es para `n=3`. Supondremos que f(n) es la función n^2.

invocación	n	valor retornado
1ª	3	$3^2+5 = 14$
2ª	2	$2^2+1 = 5$
3ª	1	$1^2+0 = 1$
4ª	0	0

La 4ª invocación retorna a la tercera el valor 0. La 3ª invocación usa ese valor para calcular el resultado y retorna a la 2ª invocación el valor 1. Igualmente, la 2ª invocación usa ese 1 para calcular el resultado y retorna a la 1ª invocación el valor 5. Finalmente, la 1ª invocación usa ese valor 5 para calcular el resultado final de 14.

Fases del diseño recursivo

Obtener una definición recursiva de la función a implementar a partir de la especificación

- Establecer caso directo
- Establecer caso recursivo

Diseñar el algoritmo con una instrucción condicional

Argumentar sobre la terminación del algoritmo

Ejemplo 2: Convertir un número decimal a otra base de numeración

Variables:

- `i`: número entero a convertir
- `base`: base destino (2≤`base`≤10)
- El resultado se retorna como un texto

Caso directo

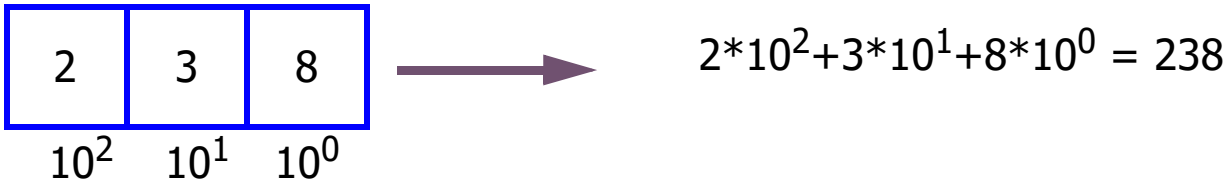
- el número solo tiene una cifra: si `i<base`, el resultado es `i`

Caso recursivo

- convertir `i//base` a la base deseada (invocando la misma función)
  - se trabaja primero con la parte más significativa
- y añadir `i%b`
  - se trabaja con la parte menos significativa al final

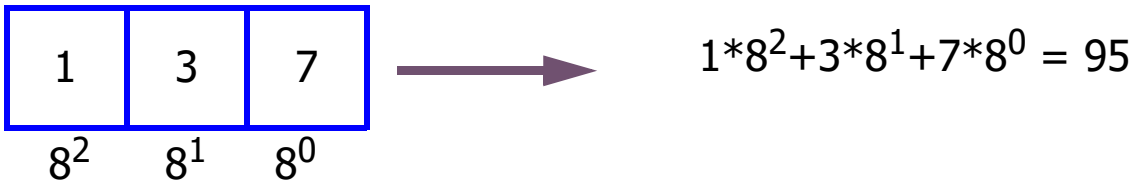
Notas:

La numeración arábiga que usamos tiene diez cifras (del 0 al 9) y para representar números más grandes usamos la notación posicional. Creamos números de varias cifras, donde la cifra de la derecha tiene un peso de 10^0=1, la siguiente cifra representa las decenas y tiene un peso de 10^1=10, la siguiente son las centenas con un peso de 10^2=100, y así sucesivamente.



En general, la cifra que ocupa la posición n desde la derecha, empezando en 0, tiene un peso de 10^n.

Si en lugar de usar 10 cifras usamos menos o más, por ejemplo 8 (base octal), el peso de la cifra n es 8^n y, en general, si la base es b, b^n. Por ejemplo, el número 137 en octal equivale en decimal a:



El algoritmo para convertir un número decimal a otra base consiste en dividirlo por la base y luego operar por separado con el cociente entero (truncado) y el resto de la división.

- El resto de la división nos da la cifra de la derecha (la menos significativa).
- Al cociente hay que aplicarle el mismo algoritmo para ir obteniendo las siguientes cifras.

Ejemplo 2 (cont.)

```
def conv_base(i: int, base: int)-> str:
    """
    Convierte un número entero a cualquier base entre 2 y 10,
    de forma recursiva

    Args:
        i: el número a convertir
        base: la base destino

    Returns:
        El string con el número en la base deseada
    """

    if i < base:
        # caso directo: el número i solo tiene una cifra
        return str(i)
    else:
        # caso recursivo: el número i tiene varias cifras
        return conv_base(i//base, base) + str(i%base)
```

Notas:

La función `conv_base()` recibe como parámetro un número decimal, `i`, y retorna un string conteniendo las cifras de ese número en la base que se pasa también como parámetro.

- Por sencillez, la base debe estar entre 2 y 10.
- Para bases superiores, se puede usar el mismo algoritmo pero habría que transformar cifras superiores a 9 en letras.

Puede observarse que el caso directo se da cuando `i` es menor que la `base` y por tanto el número solo tiene una cifra.

En el caso recursivo se retorna el resultado que se obtiene poniendo primero las cifras más significativas, obtenidas mediante la llamada recursiva a la misma función y poniendo al final la cifra menos significativa.

Consideraciones sobre los datos

Datos compartidos por todas las invocaciones del algoritmo

- atributos del objeto
- estado de otros objetos externos

Datos para los que cada invocación tiene una copia posiblemente distinta

- variables locales (internas) del algoritmo
- parámetros
- valor de retorno de la función

Notas:

En las funciones recursivas, al haber múltiples invocaciones pendientes de la función, es importante gestionar bien los datos, fijándose en cuáles están compartidos por todas las invocaciones y cuáles están replicados para cada invocación.

4.6. Descripción de algoritmos mediante pseudocódigo

Una técnica muy habitual para describir algoritmos es el pseudocódigo. Tiene como objetivos:

- descripción *sencilla*, sin los formalismos de un lenguaje de programación
- descripción *independiente del lenguaje* de programación
  - directamente traducible a código en cualquier lenguaje

El pseudocódigo contiene:

- instrucciones de control presentes en todos los lenguajes
- declaraciones de datos
- expresiones con cálculos
- acciones expresadas sin el formalismo de los lenguajes

Notas:

Observar los objetivos del pseudocódigo:

- sencillez,
- independencia del lenguaje de programación.

¡ Escribir pseudocódigo es más fácil que escribir Python !

Compartir algoritmos con otras personas escribiéndolos en pseudocódigo, los hace más fáciles de traducir a otros lenguajes.

En la asignatura:

- A veces describiremos algoritmos en pseudocódigo y deberemos reescribirlos en Python.
- Otras veces lo pediremos a la inversa: describir en pseudocódigo un algoritmo que se haya escrito y probado en Python.

Pseudocódigo: Instr. condicionales

Pseudocódigo	Python
condicional	

si condición **entonces**
  instrucciones
si no
  instrucciones
fin si
	if condicion:
	  instrucciones
	else:
	  instrucciones

condicional múltiple	

si
  condición 1**=>**
  instrucciones
  condición 2**=>**
  instrucciones
  ninguna de los anteriores=>
  instrucciones
fin si
	if condicion 1:
	  instrucciones
	elif condicion 2:
	  instrucciones
	else:
	  instrucciones

Notas:

Para el pseudocódigo la notación es muy libre y no está sujeta a la gramática estricta de los lenguajes de programación.

Aquí proponemos una notación en castellano, para usar en la asignatura.

- En textos científicos es más habitual usar una notación similar pero en inglés

Observar que cada instrucción tiene un comienzo claro y un final claro marcado por la palabra `fin` y la instrucción que se está finalizando.

El contenido de cada instrucción se remarca visualmente con el sangrado, igual que en Python y en todos los lenguajes.

Por el mismo motivo por el que conviene documentar el código fuente con comentarios, el pseudocódigo también debe documentarse.

Pseudocódigo: Instrucciones de bucle

Pseudocódigo	Python
bucle while	

mientras condición
  instrucciones
fin mientras
	while condición:
	  instrucciones

bucle for con un rango	

para i **desde** 1 **hasta** n
  instrucciones
fin para
	for i **in** range(1, n+1):
	  instrucciones

bucle for que recorre una lista	

para cada x **en** lista
  instrucciones
fin para
	for x **in** lista:
	  instrucciones

- en el pseudocódigo del bucle *para* con un rango, los valores `inicial` y `final` están *incluidos*

Notas:

Una dificultad que encontraremos al expresar bucles con rango es que en Python el valor final de un rango está excluido, mientras que en muchos lenguajes y, por consecuencia en pseudocódigo, el valor final está incluido, lo mismo que el inicial.

- Es importante fijarse en este detalle, pues un rango incorrecto puede conducir a un algoritmo erróneo.
  - Esto no es un simple fallo de estilo.

Pseudocódigo: Datos, acciones y expresiones

Declaraciones de variables; ejemplos:

```
i: entero
temperatura: real
s: texto
# lista de números reales
a: lista[real]
# el tamaño de la lista sería
longitud(a)
```

Expresiones con cálculos; ejemplo:

```
i=suma+3*x
```

Acciones expresadas sin el formalismo de los lenguajes; ejemplos:

```
leer i y j de teclado
escribir resultado en la pantalla
```

Notas:

La creación de variables se hace igual que en Python, pero usando los tipos en castellano (entero, real, texto) en lugar de usar los nombres Python.

Las expresiones con cálculos se pueden hacer en notación matemática o como en Python.

La lectura y escritura de teclado y en pantalla se hacen indicándolo en castellano.

Pseudocódigo: invocar funciones

Invocar una función

```
función(argumentos)
```

Invocar un método:

```
objeto.método(argumentos)
```

Notas:

Invocar funciones y métodos lo hacemos como en Python.

Pseudocódigo: definir funciones

Usaremos esta estructura para definir una función que no retorna nada:

```
función nombre (parámetros)
    instrucciones
fin función
```

Para una función que retorna un valor:

```
función nombre (parámetros): tipo_retornado
    instrucciones
    retorna expresión
fin función
```

Si es un método, no hace falta poner el parámetro `self`, pues esto es especial de Python

```
método nombre (parámetros) ...
```

Notas:

La definición de funciones y métodos se hace indicando el principio y el final con claridad, como en las instrucciones de control.

Las cabeceras de métodos y funciones son parecidas a las de Python,

- sustituyendo la `"->"` por el símbolo `":"`, como en los diagramas de clase,
- y omitiendo el parámetro self, que es especial de Python.

Ejemplo: suma de los 100 primeros enteros positivos

Pseudocódigo	Python

suma:entero=0
para i **desde** 1 **hasta** 100
  suma=suma+i
fin para
	suma: int = 0
	for i **in** range(1, 101):
	  suma += i

También para incrementos distintos de uno (ej: nº pares):

Pseudocódigo	Python

suma:entero=0
para i **desde** 2 **hasta** 100 **paso** 2
  suma=suma+i
fin para
	suma: int = 0
	for i **in** range(2, 101, 2):
	  suma += i

Ejemplo: recorrido de una lista de números

Recorriendo cada casilla (bucle "para cada"):

Pseudocódigo	Python

`li: lista[entero] =`
`  [2, 5, 8, 34, 56]`
 	
`para cada elem en li`
`  Mostrar elem en pantalla`
`fin para`
	`li: List[int] = \`
	`  [2, 5, 8, 34, 56]`
	
	`for elem in li:`
	`  print(elem)`

Ejemplo

Vamos a escribir una función para obtener el valor del ***logaritmo*** de `y=1+x` de acuerdo con el siguiente desarrollo en serie

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n-1}\frac{x^n}{n}, \quad -1 < x \leq 1$$

Para calcular de manera eficiente el signo y el numerador:

- no usaremos potencias
- el signo va cambiando de un término al siguiente
- el numerador siempre es el del término anterior por x

Diseño

```
# Calcula el logaritmo de y, sumando n términos
# de su desarrollo en serie
función logaritmo (y: real, n: entero): real
    x: real = y-1
    log: real = 0 # para recoger el resultado
    numerador: real = x  # primer numerador
    signo: entero = 1   # primer signo
    para i desde 1 hasta n
        log=log+signo*numerador/i
        # calculamos el numerador y el signo
        # para la próxima vez
        numerador=numerador*x
        signo=-signo
    fin para
    retorna log
fin función
```

Notas:

En este ejemplo se calcula un sumatorio, pero evitamos la operación "elevar a" sustituyéndola por sumas, restas o productos.

- Aunque en los computadores normales (por ejemplo un PC) la operación "elevar a" es bastante eficiente, en algunos computadores más básicos, la operación "elevar a" puede ser cientos de veces menos eficiente que un producto o una suma.

Creamos una variable para el signo de cada término y otra para el numerador.

- En otros casos con un denominador complicado también podremos crear una variable para él.

La variable `log` irá recogiendo la suma de todos los términos.

- Inicializamos esta variable a cero, que es el elemento neutro de la suma.

Las variables `signo` y `numerador` reciben como valores iniciales el signo y numerador del primer término.

Dentro del bucle, que se repite `n` veces:

- Calculamos un término del desarrollo usando las variables `signo` y `denominador` y se lo añadimos a la variable `log`.
- Recalculamos las variables `signo` y `denominador` para la próxima iteración.

Finalmente, retornamos el resultado obtenido.