Parte I: Programación en un lenguaje orientado a objetos

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

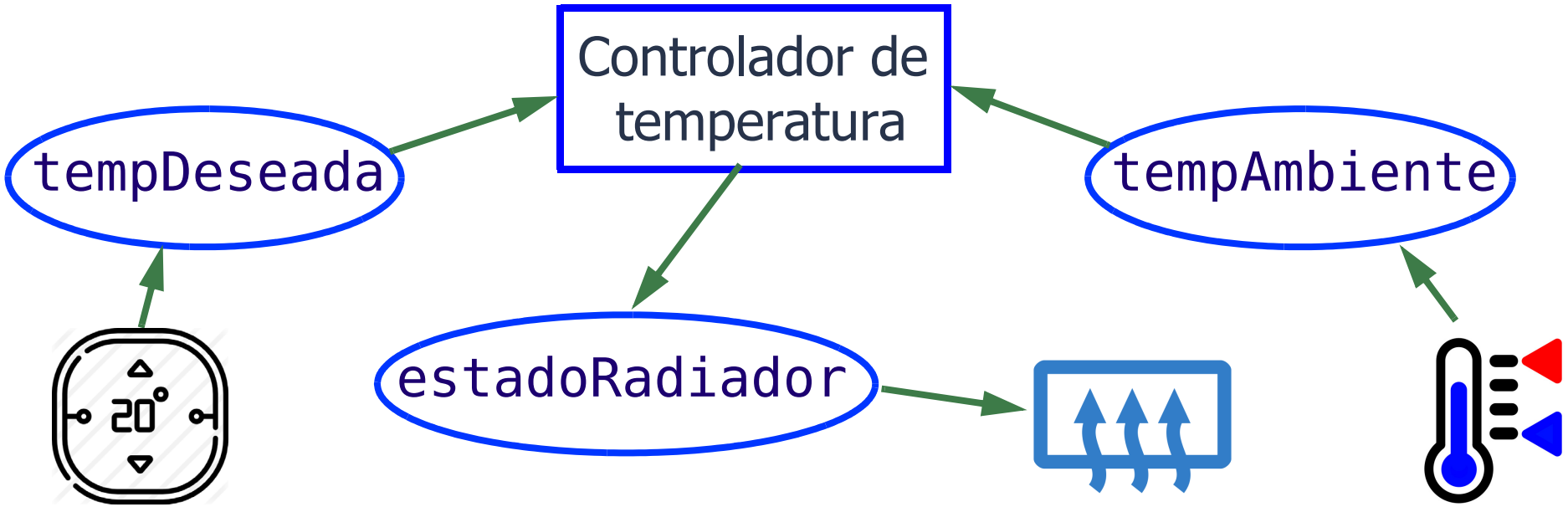
Los programas se componen habitualmente de muchas clases que contienen algoritmos, junto con datos utilizados por ellos

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

Ejemplo de algoritmo

Sistema de control de temperatura 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



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 de alto nivel:

```
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

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

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

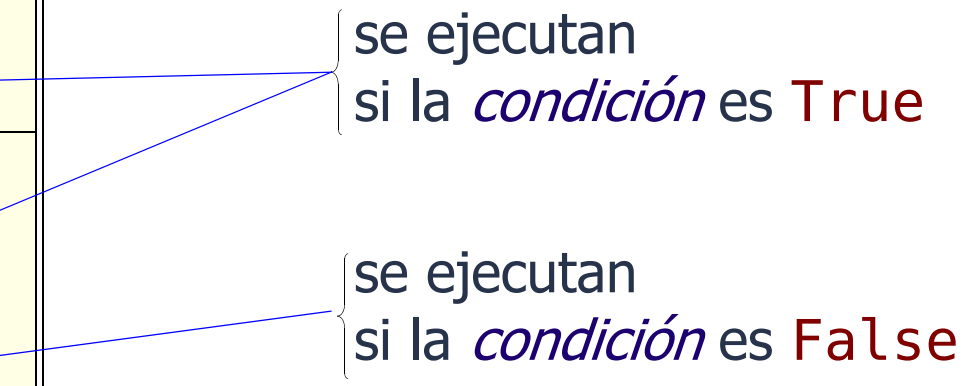
Estructura algorítmica	Descripción	Instrucción
Composición *secuencial*	Las instrucciones se ejecutan en secuencia, una tras otra	Ninguna. Simplemente se ponen las instrucciones una detrás de otra
Composición *alternativa*	En función de una condición se eligen unas instrucciones u otras	Instrucciones 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

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

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

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

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")
```

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`

Usar F8 en Spyder para detectarlo

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`

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
```

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 ▶

4.4. Instrucciones de lazo o bucle

Permiten ejecutar múltiples veces unas instrucciones

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

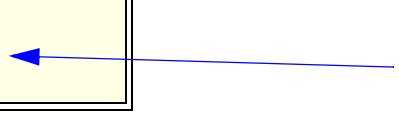
La cantidad de veces se puede establecer mediante:

- ***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`

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 (no continuamente)

Ejemplo

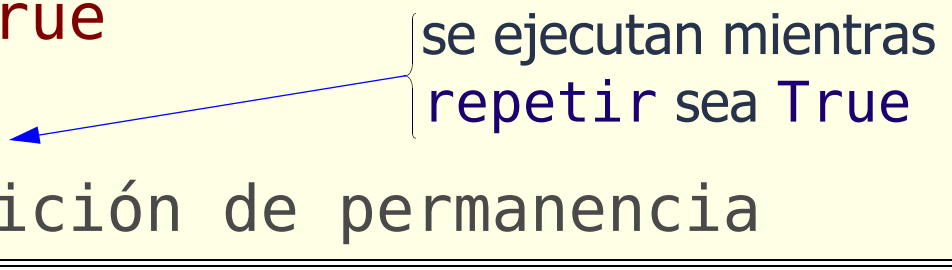
Calcular el primer entero positivo tal que la suma de él y los anteriores sea mayor que 100

```
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}"
```

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

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}")
```

Ejemplo: alternativa

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

```
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)}")
```

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

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()
```

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, ...

La variable toma sucesivamente cada valor de la secuencia

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
```

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)])
```

al final hacemos el sumatorio

expresión que se aplica a todos los elementos

de esta secuencia

Generadores (*generators*)

Cuando no necesitamos conservar la lista en memoria, en lugar de una comprensión de lista podemos usar un generador

El generador es más eficiente ya que no guarda la lista en memoria

- permite recorrer los elementos y hacer operaciones globales con ellos como sumarlos o calcular el máximo
- se escriben igual que la comprensión de lista cambiando los corchetes `[]` por paréntesis `()`

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

generador

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

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):
      ...
```

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
"""

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, seno y coseno
list_x: list[float] = list(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()
```

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

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

Ejemplo

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
    # Caso recursivo
    return f(n)+sumatorio(n-1)
```

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

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)
    # caso recursivo: el número i tiene varias cifras
    return conv_base(i//base, base) + str(i % base)
```

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

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

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 las anteriores=>
  instrucciones
fin si
	if condicion 1:
	  instrucciones
	elif condicion 2:
	  instrucciones
	else:
	  instrucciones

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*

Pseudocódigo: Datos, acciones y expresiones

Declaraciones de variables; ejemplos:

```
i: entero = 3
temperatura: real = -5.6
s: texto = "un texto"
# lista (vacía) 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
```

Pseudocódigo: invocar funciones

Invocar una función

```
función(argumentos)
```

Invocar un método:

```
objeto.método(argumentos)
```

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) ...
```

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
```