El lenguaje de programación Python3

Python3 es un lenguaje de programación con las siguientes características:

- Imperativo: los programas son sucesiones de instrucciones.
- Interpretado: se necesita de un programa especial en la máquina que lea las instrucciones y las ejecute.
- Modular: los programas se pueden dividir en varios ficheros (librerías y programa principal).
- Valores por referencia: las variables contienen la dirección de memoría del objeto al que están asignadas.
- Tipado dinámico: las variables pueden cambiar el tipo de objeto al que son asignadas durante la ejecución del programa.
- Orientado a objetos: los programas operan sobre objetos que incluyen datos y algoritmos.

A continuación describiremos un pequeño subconjunto de Python3 que nos servirá para implementar casi todos los algoritmos de la asignatura.

Las instrucciones más simples de Python3 son las siguientes:

```
# asignación simple
variable = expresion

# asignación multiple
var_1, var_2, ..., var_n = expr_1, expr_2, ..., expr_n

# condicional
if expresion:
    bloque_de_instrucciones
elif expresion:
    bloque_de_instrucciones
elif expresion:
    bloque_de_instrucciones
else:
    bloque_de_instrucciones

# bucle
while expresion:
     bloque_de_instrucciones
else:
     bloque_de_instrucciones

# no hacer nada
pass

# salir de un bucle while sin ejecutar el "else"
break

# recomenzar el bucle while
continue
```

Los bloques de instrucciones mencionados arriba corresponden con una o más instrucciones puestas una tras otra con la misma indentación. Las variables son simplemente cualquier secuencia de letras del alfabeto (mayúsculas y minúsculas), dígitos y la barra baja, pero que no empiecen con un dígito y que no correspondan con ninguna palabra reservada del lenguaje. Las expresiones pueden involucrar constantes, variables, las operaciones `+`, `-`, `*`, `/`, `//`, `%`, `**`, `&`, `|`, `^`, `~`, `<`, `>`, `==`, `!=`, `<=`, `>=`, `<<`, `>>`, `and`, `or`, `not`, e invocaciones a funciones previamente definidas. El uso de paréntesis en expresiones es necesario para cambiar el orden de precedencia predeterminado de las operaciones. En una asignación múltiple, todas las expresiones son calculadas (de izquierda a derecha) y luego asignadas a las correspondientes variables.

Python3 puede operar, entre otros, con valores booleanos (verdadero y falso), números enteros de cuaquier cantidad de dígitos, cadenas de caracteres unicode de cualquier longitud, números reales y complejos de punto flotante de precisión doble (64 bits), listas, tuplas, conjuntos y diccionarios.

Los valores booleanos son representados con las palabras `True` y `False`. Las operaciones lógicas entre valores booleanos son `and`, `or` y `not`. En un bucle `while`, la expresión (ver linea 18) debe dar como resultado `True` para que el siguiente bloque de instrucciones sea ejecutado y `False` para que el bucle termine. Por supuesto, cada vez que el bloque es ejecutado, la expresión condicional vuelve a evaluarse para determinar si se debe volver a entrar o si se debe salir del bucle. Antes de salir del bucle, el bloque de instrucciones dentro del `else` es ejecutado (si es que está presente), a menos que se haya salido a través de un `break`. De forma similar, las condicionales determinan cuál es el único bloque de instrucciones que debe ejecutarse en función de los resultados de las expresiones booleanas (ver lineas 8, 10, 12, 14). La parte `else:` puede omitirse y puede haber cualquier cantidad de `elif expresion:` como se necesite (o incluso no haber ninguna de estas).

Los enteros se escriben en notación decimal (por ejemplo, `9489284792794` es un entero válido). También es posible escribir enteros en base 2, 8 o 16, precediendo los dígitos con `0b`, `0o` o `0x`, respectivamente. Da lo mismo escribir `1234`, `0b10011010010`, `0o2322` o `0x4d2`. Las operaciones entre enteros son: suma `+`, resta `-`, multiplicación `*`, división real `/`, división entera `//`, resto de la división `%`, potenciación `**`, and bit-a-bit `&`, or bit-a-bit `|`, xor bit-a-bit `^`, not bit-a-bit `~`, comparaciones de igual `==` y de diferente `!=`, comparaciones de tipo desigualdad `<`, `>`, `<=`, `>=` y desplazamiento de bits hacia la izquierda `>>` y hacia la derecha `<<`. Las comparaciones dan como resultado un valor booleano. La división real da como resultado un número de punto flotante y la división entera da el cociente redondeado hacia abajo. La potenciación da un entero si el exponente es positivo o cero y un número de punto flotante si el exponente es negativo. El desplazamiento de bits hacia la derecha es equivalente a multiplicar por una potencia de dos, es decir, `1 << 20` da como resultado 2^{20}. El desplazamiento hacia la izquierda corresponde con la división entera por una potencia de dos, es decir, `83 >> 3` dará como resultado 10, ya que $\lfloor 83/2^3 \rfloor = 10$.

Detalle técnico: Los números enteros tienen asociado una secuencia infinita (hacia la izquierda) de bits llamada *complemento a dos*. Para un $n \geq 0$, esta secuencia es simplemente la escritura binaria de n con ceros a la izquierda. Por ejemplo, el entero `n = 0b10111 = 23`, tiene asociada la sucesión $\cdots 000010111$. El conjunto de estas sucesiones infinitas de bits forma un anillo conmutativo $\mathbb{Z}_2$ (con la suma y multiplicación calculadas como habitualmente, es decir, teniendo en cuenta los acarreos). Podría pensarse que $\mathbb{Z}_2$ es el conjunto de enteros con *infinitos dígitos binarios*. En este anillo, el inverso aditivo de $\cdots 000001$ es la sucesión $\cdots 111111$ de todos unos, ya que al hacer la adición (con acarreo) se obtendrían todos ceros. Un entero $n < 0$ tendrá entonces asociada la secuencia que es el inverso aditivo de la de $-n$ en el anillo $\mathbb{Z}_2$. Por ejemplo, el `n = -0b10100 = -20` tendrá asociada la secuencia $\cdots 1111101100$. La aplicación $\mathbb{Z} \to \mathbb{Z}_2$ que manda cada entero a su representación en complemento a dos, es un morfismo de anillos.

Las operaciones bit-a-bit `&`, `|`, `^` y `~` trabajan con la representación complemento a dos de los argumentos. Por ejemplo, si `a = 0b10011 = 19` y `b = 0b10101 = 21`, entonces los resultados de las operaciones bit-a-bit serían `a&b = 0b10001 = 17`, `a|b = 0b10111 = 23`, `a^b = 0b110 = 6` y `~a = -0b10100 = -20`.

```
1 # este programa calcula 10000! y lo imprime
2 n = 1
3 m = 1
4 while n <= 10000:
5     m = m * n
6     n = n + 1
7 print(m)
```

Programa 1: `factorial_10000.py`

La linea 5 del programa 1 puede abreviarse escribiendo `m *= n` y de forma similar, la linea 6 puede escribirse `n += 1`. Estas formas son muy frecuentes en Python3. La estructura del bucle while de las lineas 4–6 es también muy común: la variable `n` actua como un *contador* que va incrementandose en una unidad en cada iteración. El efecto es que la linea 5 se ejecuta 10000 veces, cada vez con un valor de $n = 1, 2, \ldots, 10000$ diferente. La variable m actua como un *acumulador*, en el que se van multiplicando todos los valores de n. La función predefinida `print(x)` imprime el objeto `x` en el dispositivo de salida (usualmente el monitor del ordenador).

A partir de la versión 3.10.7 del intérprete de Python3, las conversiones entre enteros y cadenas de caracteres están limitadas a un máximo de 4300 dígitos decimales. Para que el programa anterior funcione, se invocar al intérprete con la variable de entorno `PYTHONINTMAXSTRDIGITS=0`. Si no hicieramos esto, nos daría un error en la linea 7, ya que 10000! tiene 35660 dígitos.

```
1 # este programa imprime la suma de los dígitos decimales de 3^10000
2 a = 3 ** 10000
3 s = 0
4 while a > 0:
5     s = s + a % 10        # a % 10 es el dígito de las unidades de a
6     a = a // 10           # división entera de a por 10
7 print(s)
```

Programa 2: `suma_digitos_3pow10000.py`

Las lineas 5 y 6 del programa 2 podrían escribirse como `s += a % 10` y `a //= 10`.

Los valores reales de punto flotante también se escriben en notación decimal, pero deben incluir el punto en algún lugar para diferenciarlos de los enteros (por ejemplo, `12.9287492` o `1.0`). También pueden estar escritos en notación científica, usando la letra `e` para separar la mantisa del exponente (por ejemplo, `3.234e-29` significa $3.234 \cdot 10^{-29}$). Para indicar la parte imaginaria de un número complejo se añade una letra `j` al final del número sin dejar ningún espacio (por ejemplo, `3.0+5e-3j` significa $3 + 5 \cdot 10^{-3}i$). Exceptuando las operaciones de desplazamiento de bits, todas las demás operaciones pueden trabajar con valores reales y complejos.

```
1  # aproximación del área de un cuarto de círculo de radio 1 haciendo la
2  # suma de Riemann correspondiente a la integral de raiz(1-x^2) entre 0
3  # y 1, dividiendo al intervalo de integración en 100000 subintervalos.
4  area = 0.0
5  i = 0
6  while i < 100000:
7     x = i/100000               # extremo izquierdo del intervalo
8     y = (1 - x**2) ** 0.5      # altura del rectangulo
9     area += y/100000           # acumulamos el área
10    i += 1
11 print(area)                   # el valor exacto debería ser pi/4
```

Programa 3: `pi_div_4_aprox.py`

Las cadenas de caracteres se escriben entre comillas simples o dobles (por ejemplo, `'hola a todos!'`, `"300 euros"`). Las únicas operaciones permitidas con cadenas de caracteres son la concatenación `+`, la repetición `*` y las comparaciones `==`, `!=`, `<`, `>`, `<=`, `>=`. Se puede obtener la subcadena los caracteres entre los indices i y j de una cadena dada, usando el operador `[i:j]`. El primer caracter de una cadena tiene índice 0 y el operador no incluye el caracter con índice j en la subcadena. Por ejemplo, si `a='python'`, la operación `a[1:4]` dará como resultado `'yth'`. En caso de que i no esté explicitamente dado, se entenderá que $i = 0$ y de forma similar, si j no está dado, se supone que j es la longitud de la cadena. Siguiendo con el ejemplo, `a[:3]` producirá `'pyt'` y `a[3:]` dará `'hon'`. Los índices negativos indican una posición empezando desde la derecha, es decir, en nuestro ejemplo, tendríamos que `a[-1]` es `'n'` y `a[-3:-1]` es `'ho'`. En el caso de que $i < 0$ y j no esté dado, se entenderá que $j = 0$, es decir, `a[-4:]` es `'thon'`. De forma similar, si $j < 0$ pero i no está dado, i será implícitamente el negativo de la longitud de la cadena.

La función predefinida `len(s)` devuelve la longitud de la cadena `s` y la función predefinida `str(x)` convierte un objeto `x` a una cadena de caracteres.

```
1  # parte 1: obtenemos la representación binaria de 3^10000 y su longitud
2  n = 3 ** 10000
3  s = ""                  # empezamos con una cadena de caracteres vacía
4  while n > 0:
5      s = str(n%2) + s    # n%2 es el bit de las unidades de n
6      n = n >> 1          # dividimos por 2 (ignorando los decimales)
7  print("representación binaria =", s)
8  print("número de bits =", len(s))
9
10 # parte 2: buscamos si la secuencia 101010101010 aparece en s o no
11 i = 0
12 encontrada = False
13 while i < len(s)-12 and not encontrada:
14     if s[i:i+12] == "101010101010":
15         encontrada = True
16     i = i + 1
17 if encontrada:
18     print("secuencia 101010101010 encontrada en la posición", i)
19 else:
20     print("secuencia 101010101010 no encontrada")
```

Programa 4: `bin_3pow10000.py`

Las funciones predefinidas `bool(x)`, `int(x)` y `float(x)` convierten, si es posible, un objeto `x` a un booleano, entero y a un número real de punto flotante, respectivamente. Usualmente son invocadas con `x` una cadena de caracteres y por lo tanto son, en ese sentido, operaciones inversas a `str(x)`.

Las funciones `bin(x)`, `oct(x)` y `hex(x)` toman un valor entero `x` y devuelven su representación binaria, octal y hexadecimal, respectivamente (con el correspondiente prefijo `0b`, `0o` y `0x`). Por ejemplo, la parte 1 del programa 4 (lineas 1–6) podría sustituirse por `s = bin(3**10000)[2:]`.

Cada caracter está internamente representado en Python3 por su valor Unicode. La función `chr(n)` toma un entero `n` y devuelve una cadena de caracteres de longitud 1 con el correspondiente caracter Unicode. La función `ord(s)` hace exactamente lo opuesto, es decir, toma un cadena de un solo caracter y devuelve el valor numérico Unicode de ese caracter. Por ejemplo, `chr(241)` devuelve `'ñ'` y `ord('@')` da `64`.

En Python3 es posible definir funciones nuevas con el comando `def`, tal como se muestra a continuación:

```
1  def nombre_de_la_funcion(arg1, arg2, ...):
2      bloque_de_instrucciones
```

La instrucción `return expr1, expr2, ...` se utiliza dentro de la definición de una función para indicar cuál es el resultado de la misma.

El siguiente ejemplo, muestra una función que determina si un entero $n \geq 2$ dado es primo o no (la función devuelve un valor booleano) y luego la invoca en un bucle while para listar todos los primos entre 2 y 100.

```
1  # este programa imprime todos los números primos entre 2 y 100
2  def es_primo(n):           # esta función determina si un n >= 2 es primo
3      d = 2                  # buscando un divisor d entre 2 y n-1
4      while d < n:
5          if n%d == 0:       # resto = 0 significa que d divide a n
6              return False   # y por lo tanto n no es primo
7          d = d + 1
8      return True
9
10 n = 2
11 while n <= 100:
12     if es_primo(n):
13         print(n)
14     n = n + 1
```

Programa 5: `primos_hasta_100.py`

Puede verse que la función `es_primo`(n) en el programa 3 es bastante ineficiente. Puede mejorarse bastante si en lugar de buscar divisores entre 2 y $n-1$ se los busca entre 2 y $\lfloor\sqrt{n}\rfloor$. Eso podría conseguirse simplemente cambiando la linea 4 por `while d*d <= n:` Otra posible mejora sería determinar primero si n es par o no y luego sólo comprobar los `d` impares.

Las variables `n` y `d` de las lineas 2–8 son *locales* a la función `es_primo`. Podrían cambiarse todas las `n` en las lineas 10–14 por `d` y el programa funcionaría correctamente. La asignación `d = 2` que sucede dentro de la función (linea 3) no afectaría a la variable `d` externa.

Hasta ahora hemos visto que Python3 permite operar con valores booleanos, enteros de precisión arbitraria, números reales y complejos de doble precisión (punto flotante) y cadenas de caracteres.

Python3 también permite operar con colecciones de objetos, en forma de listas, tuplas, conjuntos y diccionarios.

Una *tupla* es simplemente una secuencia finita de objetos, indicados entre paréntesis. Por ejemplo, `(2,4,True,(5,1.3,"hola!"))` es una tupla de longitud 4, formada por dos enteros, un booleano y una tupla. La tupla de longitud 0 se representa `()` y las de longitud 1 con una coma adicional luego de la única entrada, como por ejemplo `("álgebra",)`, para que no haya ambigüedad con el uso habitual de los paréntesis en expresiones. Al igual que con las cadenas de caracteres, las operaciones permitidas son la concatenación `+`, la repetición `*`, las comparaciones `==`, `!=`, `<`, `>`, `<=`, `>=`, y la extracción de una subtupla usando `[i:j]`. Hay una nueva operación `in` que permite determinar si un objeto `x` está o no en una tupla. Por ejemplo, si `t = (1,3,"aaa")`, la expresión `2 in t` dará como resultado `False`.

Las *listas* son también secuencias finitas de objetos, pero en este caso, se usan corchetes en lugar de paréntesis para indicarlas. Por ejemplo, `[2,False,(3,5.9),"hola!",[-2,8]]` es una lista de longitud 5. La lista vacía se representa `[]` y las de un sólo elemento no necesitan la coma adicional, como por ejemplo `[2021]`. Acepta las mismas operaciones que las tuplas. Es posible convertir una tupla `x` en una lista, con la función predefinida `list(x)`, y convertir una lista `x` en una tupla con la función predefinida `tuple(x)`.

```
# este programa obtiene la lista de puntos (x,y)
# que satisfacen 0<=x,y<13 y 13|y^2-x^3-7.
puntos = []                                 # empezamos con la lista vacía
x = 0                                       # bucle con x=0,1,...,12
while x < 13:
    y = 0                                   # bucle con y=0,1,...,12
    while y < 13:
        if (y**2-x**3-7)%13 == 0:           # si el punto cumple la condición
            puntos += [(x,y)]               # lo agregamos a la lista
        y += 1
    x += 1
print(puntos)
```

Programa 6: `curva_eliptica_mod13.py`

El programa 6 imprime una lista de 6 tuplas de dos coordenadas cada una. El programa también funcionaría si se cambia la linea 3 por `l = ()` y la linea 9 por `l += ((x,y),)`, pero en este caso imprimiría una tupla de 6 tuplas.

Detalle técnico: Con lo que hemos visto hasta ahora, parecería que las listas y las tuplas son completamente intercambiables. Sin embargo, hay una sutil diferencia entre ambas: es posible modificar, agregar o borrar elementos dentro una lista, pero no de una tupla. Se dice que las listas son objetos mutables y las tuplas inmutables. Para conseguir modificar un objeto dentro de una lista `l`, basta con asignarle un valor. Por ejemplo, `l[4] = "xxx"` cambia la componente de índice 4 de `l` por la cadena de caracteres `"xxx"`. Esta acción no crea una nueva lista, sino que modifica la ya existente. Para borrar la componente de índice `i` de una lista `l` se usa la instrucción `del l[i]`. Para reemplazar parte de una lista `l1` por otra `l2`, se usa la asignación `l1[i:j] = l2`. Para añadir un objeto `x` al final de una lista `l` se utiliza `list.append(l, x)`, o equivalentemente, `l.append(x)`.

```
# este programa es para mostrar la mutabilidad de las listas

# la siguiente función ordena una lista l de enteros de menor a mayor
def ordenar(l):
    n = len(l)
    i = 0                                   # bucle i = 0,1,...,n-1
    while i < n:
        j = i+1                             # bucle j = i+1,...,n-1
        while j < n:
            if l[i] > l[j]:                 # si estan mal ordenados,
                l[i], l[j] = l[j], l[i]     # los cambiamos de lugar
            j += 1
        i += 1

# ahora creamos una lista lst1
lst1 = [2,5,-1,3,77,8,-3]
print(lst1)

# la lista lst2 es la MISMA que lst1 y no una simple copia de lst1
# las asignaciones en Python3 copian REFERENCIAS (direcciones de
# memoria) y no crean nuevos objetos
lst2 = lst1
print(lst2)

# agregamos un elemento a lst1
lst1.append(-17)
print(lst1)

# notar que lst2 TAMBIÉN CAMBIA!
print(lst2)
```

```
31
32 # del mismo modo, al invocar la función ordenar, la variable l será
33 # la MISMA lista lst1, y por eso, todas las acciones hechas sobre l
34 # están hechas sobre lst1 (y también sobre lst2):
35 ordenar(lst1)
36 print(lst1)
37 print(lst2)
```

Programa 7: `ordenar_lista.py`

Los *diccionarios* son secuencias de objetos indexadas por otros objetos. Estos se representan usando llaves, del siguiente modo: {`indice:objeto, indice:objeto, ...`}. Por ejemplo, la asignación `d = {"hola":"hello", "perro":"dog", "pi":3.14}` define un diccionario `d` de tres objetos indexados por cadenas de caracteres. El valor de `d["hola"]` es `"hello"` y el de `d["pi"]` es `3.14`. El diccionario vacío se representa {}. Los diccionarios son objetos mutables, por lo que es posible cambiar, borrar o añadir traducciones. En el ejemplo, si se hace `d["gato"] = "cat"` se estaría añadiendo una nueva traducción, si se hace `d["pi"] = 3.141592` se modificaría la traducción existente, y por último, si se hace `del d["hola"]` se borraría esa traducción del diccionario. Las únicas operaciones permitidas son `in`, `==` y `!=`.

```
1  dicc = {"a" : ".-", "b" : "-...", "c" : "-.-.", "d" : "-..", "e" : ".",
2  "f" : "..-.", "g" : "--.", "h" : "....", "i" : "..", "j" : ".---", "k" :
3  "-.-", "l" : ".-..", "m" : "--", "n" : "-.", "o" : "---", "p" : ".--.",
4  "q" : "--.-", "r" : ".-.", "s" : "...", "t" : "-", "u" : "..-", "v" :
5  "...-", "w" : ".--", "x" : "-..-", "y" : "-.--", "z" : "--.." }
6
7  texto = "este es un texto de prueba para probar el diccionario"
8
9  n = len(texto)
10 i = 0
11 morse = ""
12 while i < n:
13     letra = texto[i]
14     if i != 0:
15         morse += " "             # separador entre letras es " "
16     if letra == " ":
17         morse += " "             # separador entre palabras es "   "
18     else:
19         morse += dicc[letra]
20     i += 1
21
22 print(morse)
```

Programa 8: `codigo_morse.py`

Detalle técnico: Debido a la forma en la que los diccionarios están implementados (por razones de eficiencia), los índices deben ser objetos *hashables*. Si bien es difícil definir exactamente en este punto cuales son esos, podríamos decir que cualquier objeto booleano, entero, real, complejo y cadena de caracteres es hashable, como así también las tuplas cuyas componentes sean hashables. Los diccionarios y las listas no son hashables.

Los *conjuntos* son colecciones de objetos distintos (que deben ser hashables) y se indican usando llaves {`obj1, obj2, obj3, ...`}. Las repeticiones se consideran como un único elemento. Las operaciones entre conjuntos son: pertenencia `in`, unión `|`, intersección `&`, diferencia `-` y diferencia simétrica `^`. La función predefinida `set(x)` convierte un objeto `x` en un conjunto. Si se la invoca con una lista o una tupla, devuelve el conjunto de sus elementos sin repetir. El conjunto vacío no puede ser indicado con {}, ya que eso representa el diccionario vacío, por lo que se debe utilizar `set()`. Los conjuntos son

objetos mutables, y por lo tanto, se les puede añadir o quitar elementos. Para añadir un objeto `x` a un conjunto `s` se usa `s.add(x)` y para quitarlo se usa `s.discard(x)`. Los conjuntos no son hashables.

```
1  # queremos determinar cuántos de los 100 posibles números entre 00 y
2  # 99 aparecen como los dos últimos dígitos de los números 3^i para
3  # i=1,..,5000
4
5  # generamos los 5000 numeros (3^i mod 100) y los vamos agregando al
6  # conjunto nums
7  nums = set()                    # empezamos con nums = conjunto vacío
8  i = 1
9  while i <= 5000:
10    nums |= {(3 ** i) % 100}    # la | es la union de conjuntos
11    i += 1
12
13 # imprimimos el conjunto nums y su cantidad de elementos
14 print(nums)
15 print(len(nums))
```

Programa 9: `ultimos_digitos_3_pow_i.py`

En el programa 9, es importante usar conjuntos y no tuplas o listas, ya que no nos interesan los elementos repetidos. La linea 10 podría cambiarse por `nums.add((3 ** i) % 100)` para evitar crear un nuevo conjunto en cada iteración.

Detalle técnico: El equivalente inmutable (y hashable) de los conjuntos son los conjuntos congelados, que se indican `frozenset({obj1, obj2, ...})`. Permiten las mismas operaciones que los conjuntos, excepto añadir y quitar elementos. El conjunto congelado vacío se indica `frozenset()`.

En el programa 9 podríamos cambiar la linea 7 por `nums = frozenset()` y la linea 10 por `nums |= frozenset({(3 ** i) % 100})` ya que no estamos utilizando la mutabilidad en ninguna parte. Cada vez que se ejecuta la linea 10, se crearía un nuevo conjunto congelado. Lógicamente, en este caso no podríamos cambiar la linea 10 por `nums.add((3 ** i) % 100)`.

Hasta ahora hemos trabajado con bucles while, que permiten repetir un bloque de instrucciones mientras una cierta condición se mantenga verdadera. Python3 también tiene los bucles `for`:

```
1 for variable in objeto_iterable:
2     bloque_de_instrucciones
3 else:
4     bloque_de_instrucciones
```

que repiten el bloque de instrucciones con la variable recorriendo todos los valores en el objeto iterable (cadena de caracteres, lista, tupla, conjunto, conjunto congelado o diccionario). Para el caso de diccionarios, se pueden usar dos variables:

```
1 for var_1, var_2 in diccionario:
2     bloque_de_instrucciones
3 else:
4     bloque_de_instrucciones
```

la primera para el índice y la segunda para el valor. Si se usa una sola, el bucle sólo recorre los índices. **Cuidado:** No se debe modificar el objeto iterable usado en el bucle dentro del bloque de instrucciones.

Los bucles `for` pueden opcionalmente incluir un `else`. En ese caso, una vez que el bucle acaba normalmente (es decir, no con un `break`), se ejecutan las instrucciones dentro del `else`.

Python3 tiene un objeto iterable `range(n)` (que se puede usar en un bucle `for`) que representa todos los números entre 0 y $n-1$. Es decir, `for i in range(10):` repite el siguiente bloque de instrucciones para $i = 0, 1, \ldots, 9$. Si se escribe `range(n,m)`, se obtienen los números entre `n` y `m-1`.

```python
# este programa determina de cuántas formas se puede escribir el
# número 10000 como suma de dos primos

def es_primo(n):
    for d in range(2,n):
        if n%d == 0:
            return False
    return True

# obtenemos el conjunto de primos <= 10000
primos = set()                          # conjunto vacío
for p in range(2,10001):                # p = 2,3,...,10000
    if es_primo(p):                     # si p es primo
        primos |= {p}                   # lo agregamos al conjunto

# obtenemos el conjunto de numeros de la forma 10000-p con p primo
restas = set()
for p in primos:
    restas |= {10000-p}

print(len(primos & restas))  # el & es la intersección de conjuntos
```

Programa 10: `goldbach_10000.py`

Otro aspecto importante de Python3 es la modularidad. Un programa grande puede ser dividido en varios ficheros (librerías) para una mejor organización. Desde el programa principal, el comando `import nombre_de_la_libreria`, ejecuta el programa `nombre_de_la_libreria.py` en una nueva instancia del intérprete, y da acceso a todas las variables (o funciones) definidas en del mismo a través de la notación `nombre_de_la_libreria.nombre_de_la_variable`. En el caso de que la librería tenga un nombre que colisione con otro ya definido en el programa que la quiere importar, o que simplemente se quiera usar otro nombre, se puede utilizar el comando `import nombre_de_la_libreria as otro_nombre`. Si se hace esto, se accederá a las variable (o funciones) a través de la notación `otro_nombre.nombre_de_la_variable`.

```python
# este es un ejemplo de librería en la que se definen cuatro funciones:
# raiz_cuadrada, es_primo, primo_siguiente y k_esimo_primo_siguiente.

# esta función calcula la parte entera de la raíz cuadrada de un entero
# x >= 0, utilizando una búsqueda binaria.
def raiz_cuadrada(x):
    izq = 0
    der = x+1
    while izq < der-1:                  # buscamos en [izq,der)
        med = (izq+der)//2
        if med*med <= x:
            izq = med
        else:
            der = med
    return izq

# esta función determina la primalidad de n >= 2 comprobando si tiene
# o no divisores en el intervalo [2, raiz_cuadrada(n)].
def es_primo(n):
    r = raiz_cuadrada(n)
    d = 2
    while d <= r:
        if n%d == 0:
            return False
        d += 1
```

```
26      return True
27
28  # esta función devuelve el menor primo mayor que un n >= 1 dado.
29  def menor_primo_mayor_que(n):
30      n += 1
31      while not es_primo(n):
32          n += 1
33      return n
34
35  # esta función devuelve el k-ésimo primo mayor que un n >= 1 dado
36  # para cualquier k >= 1.
37  def k_esimo_primo_mayor_que(n, k):
38      for i in range(k):
39          n = menor_primo_mayor_que(n)
40      return n
```

Programa 11: `libreria_ejemplo.py`

```
1  # este programa utiliza la librería "libreria_ejemplo.py" que
2  # escribimos previamente.
3
4  # importamos la librería y le damos el nombre "le".
5  import libreria_ejemplo as le
6
7  # calculamos el sexto primo a partir de 10000.
8  print (le.k_esimo_primo_mayor_que(10000, 6))
```

Programa 12: `programa_ejemplo.py`

Se considera buen estilo de programación el separar tareas en pequeñas funciones y el dividir un programa en varias librerías bien estructuradas.

La función `raiz_cuadrada` definida en `libreria_ejemplo.py` es muy eficiente. Se puede comprobar que el bucle while se ejecutará aproximadamente $\log_2(x)$ veces. Sin embargo, la función `es_primo` no es nada eficiente, ya que el bucle principal se ejecutará aproximadamente $\sqrt{n}$ veces, lo que puede ser muy grande.

Problema 1.1. Si escribimos todos los números naturales menores que 10 que son múltiplos de 3 o de 5, obtenemos 3, 5, 6 y 9. La suma de esos números es 23. Calcular la suma de todos los múltiplos de 3 o 5 menores que 1000.

Problema 1.2. Cada término de la sucesión de Fibonacci es la suma de los dos anteriores. Los dos primeros términos son 0 y 1, así que la sucesión empieza:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Calcular la suma de los números pares menores que cuatro millones que aparecen entre los términos de la sucesión de Fibonacci.

Problema 1.3. Si se ordenan los primos en orden creciente se obtiene $2, 3, 5, 7, 11, 13, \ldots$. ¿Cuál es el primo que ocupa el lugar 10001 de la sucesión?

Problema 1.4. La conjetura de Collatz afirma que si empieza por cualquier número entero $n \geq 1$ y se lo va cambiando iterativamente de acuerdo con la regla

$$\begin{aligned} n \text{ par} &\longrightarrow n/2 \\ n \text{ impar} &\longrightarrow 3n+1 \end{aligned}$$

se llega siempre a 1. Verificar que la conjetura es cierta para todo n menor que 1000000 y determinar cuál de esos valores de n produce la cadena más larga hasta llegar al 1.

Problema 1.5. La constante de Champernowne es el número irracional que se obtiene al concatenar todos los números naturales, del siguiente modo:

$$0.12345678910\underline{1}11213141516171819202122 23\ldots$$

Se puede ver que el dígito en la posición número 12 despues del punto decimal es 1 (subrayado en la linea de arriba). Sea d_n el dígito que ocupa el lugar n-ésimo. Calcular

$$d_1 \cdot d_{10} \cdot d_{100} \cdot d_{1000} \cdot d_{10000} \cdot d_{100000} \cdot d_{1000000}.$$

Problema 1.6. Implementar la función `cambiar5por3(n)` que tome un número entero n y devuelva el entero que se obtiene al cambiar cada dígito 5 que aparece en la representación decimal de n por un 3. Por ejemplo, la llamada `cambiar5por3(-5152)` debe devolver `-3132`.

Problema 1.7. Implementar la función `dia_de_la_semana(d,m,a)` que devuelva el día de la semana correspondiente a día d del mes m del año a. Se puede asumir que la fecha con la que se invoca a la función es válida y pertenece al calendario gregoriano. La función deberá devolver 0 para lunes, 1 para martes, etc.

Problema 1.8 (Atbash, Rot13, Caesar)**.** El criptosistema *atbash* consiste en cifrar una cadena de caracteres, que solo contiene letras del alfabeto inglés, cambiando cada letra por la que está en la posición opuesta en el alfabeto, es decir, la `"a"` cambia por la `"z"`, la `"b"` por la `"y"`, etc. Implementar la función `atbash(w)` que tome una cadena de caracteres `w` y la devuelva cifrada. Otra variante de este criptosistema, llamada *rot13*, cambia cada letra por la que está 13 posiciones mas adelante en el alfabeto (entendido cíclicamente). Implementarlo en la función `rot13(w)`. Más generalmente, podría cambiarse el 13 por un número $0 \leq k \leq 25$ (la clave secreta) y así tener un poco mas de seguridad (cifrado *caesar*). Implementar esta variación en la función `caesar(w,k)`. ¿Cuál es la función de descifrado?

Problema 1.9 (Vigenère)**.** En el siglo XVI, Blaise de Vigenère propuso una mejora sustancial sobre el cifrado caesar, que recién pudo ser quebrado 300 años después, lo que le dió la reputación de *le chiffrage indéchiffrable*. La idea consistía en utilizar una palabra `s` razonablemente larga como clave secreta y cifrar el mensaje `m` cambiando cada `m[i]` por el caracter que está en el alfabeto en la posición que se obtiene de sumar las posiciones que ocupan `m[i]` y `s[i]`, entendiendose que la `"a"` está en la posición 0, la `"b"` en la 1, etc. Como habitualmente la cadena `m` será mucho mas larga que la cadena `s`, a esta última se la repetirá tantas veces como sea necesario para alcanzar la longitud de `m`. Implementar este cifrado en la función `enc_vigenere(m,s)`. Por ejemplo, la llamada `enc_vigenere("attackatdawn", "lemon")` debería devolver la cadena `"lxfopvefrnhr"`. Implementar también la correspondiente función de descifrado `dec_vigenere(c,s)`.

Problema 1.10. Si se ordenan todas la palabras de 4 o menos letras (elegidas del alfabeto inglés de 26 letras) con el orden del diccionario, la primera sería la `"a"`, la segunda la `"aa"`, y la última, en la posición $26+26^2+26^3+26^4$, la `"zzzz"`. ¿Cuál es la palabra en la posición 138096? Más generalmente, implementar la función `number2word(p,n)` que, dados $1 \leq p \leq 26 + 26^2 + \cdots + 26^n$, determine la palabra en la posición p de entre todas la de n o menos letras. Implementar también la operación inversa, es decir, la función `word2number(w,n)` que tome una cadena de caracteres w de longitud entre 1 y n formada solo por letras del alfabeto inglés y devuelva su posición entre todas las de hasta n letras.

Problema 1.11. Demostrar que si $k \geq 1$ y $(k+1)! > 10^m$, entonces

$$0 \leq 10^m e - \sum_{i=0}^{k} \left\lfloor \frac{10^m}{i!} \right\rfloor < k,$$

y utilizar eso para implementar la función `digitos_de_e(n)` que imprima el valor de e con $n \geq 0$ dígitos decimales correctos. Por ejemplo, `digitos_de_e(4)` deberá imprimir 2.7182.

Sugerencia: Para una implementación eficiente, demostrar que `a//(b*c) == (a//b)//c` para todos $a, b, c \in \mathbb{N}$.

Problema 1.12. Implementar la función `factorizar(n)` que toma un número entero positivo `n` y devuelve una lista de pares ordenados de los primos que aparecen en la factorización de `n` (en orden creciente) y sus correspondientes multiplicidades. Por ejemplo, el resultado de `factorizar(1400)` debe ser `[(2,3),(5,2),(7,1)]`.

Problema 1.13. Un polinomio $f = a_0 + a_1 x + \cdots + a_d x^d \in \mathbb{C}[x]$ puede ser representado en Python3 por la lista de coeficientes `[a_0,a_1,...,a_d]`. Implementar la función `evaluar_polinomio(f,t)` que tome una lista `f` representando a un polinomio y un número complejo `t` y que devuelva $f(t)$. Por ejemplo, `evaluar_polinomio([-1.0,1-2j,2.5],1j)` debería devolver (aproximadamente) `-1.5+1j`.

Problema 1.14. Hay ocho posibles monedas de euro: las de 1, 2, 5, 10, 20 y 50 céntimos y las de 1 y 2 euros. ¿Cuántas posibilidades hay para la cantidad total de dinero que puede tener una persona con exactamente 12 monedas de euro? ¿Cuál es la menor cantidad de céntimos (a partir de 12) que no pueden pagarse con exactamente 12 monedas de euro?

Problema 1.15. Se tienen inicialmente 100 cajas (colocadas en círculo) con una bola en cada una. En cada turno, se extraen todas las bolas de una caja y se las coloca, una a una, en las siguientes cajas avanzando en sentido horario. El siguiente turno comienza en la caja donde se colocó la última bola del paso anterior. ¿En qué turno se repite la configuración inicial por primera vez?

Problema 1.16. La sucesión de Golomb $g(1), g(2), g(3), \ldots$ es la única sucesión no decreciente de números naturales en la que todo $n \geq 1$ aparece exactamente $g(n)$ veces. Es fácil ver que los primeros términos de la sucesión son 1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6,... Calcular $g(1000000)$.

Problema 1.17. La criba de Eratostenes es un método para obtener todos los números primos entre 2 y n. Se empieza con la lista de todos los números $2, 3, \ldots, n$ e iterativamente se van marcando los números como primos o compuestos siguiendo la siguiente regla: en cada iteración, se marca como primo al menor número de la lista que aún no ha sido marcado (ni primo ni compuesto) y se marcan como compuestos a todos sus múltiplos. Por ejemplo, en la primera iteración, quedará marcado el 2 como primo y los números $4, 6, 8, \ldots$ como compuestos. En la segunda iteración, se marcará al 3 como primo y a $3, 6, 9, \ldots$ como compuestos. En la siguiente iteración, el menor número aún no marcado es el 5, que será marcado como primo. Implementar ese algoritmo como la función `eratostenes(n)` que toma un $n \geq 2$ y devuelve una lista l de longitud $n + 1$ tal que $l[k] = 1$ si k es primo y $l[k] = 0$ si $k = 0$, $k = 1$ o k es compuesto.

Problema 1.18 (Conway's game of life)**.** El "juego de la vida" es un *autómata celular* diseñado por el matemático británico John Horton Conway en 1970 que simula la evolución de unos individuos ficticios que habitan una mundo rectangular dividido en $m \times n$ celdas. Cada celda puede tener como máximo un individuo. La evolución del juego está completamente determinada por el estado inicial de acuerdo con las siguientes reglas:

- Un individuo seguirá vivo en la siguiente generación si y solo si está rodeado por dos o tres individuos. Cuando un individuo muere, la celda que ocupaba queda vacía.
- Las celdas vacías rodeadas por exactamente tres individuos tendrán un individuo en la siguiente generación y en los demás casos seguirán vacías.

Representaremos al estado del juego por medio de una lista l de m listas de n ceros y unos. Los ceros indican las celdas vacías y los unos las ocupadas por individuos. Implementar la función `conway(l,m,n,k)` que devuelva el estado correspondiente a la generación k-ésima partiendo de l.

Problema 1.19. Dada una lista de enteros l de longitud $n \geq 1$, encontrar $0 \leq i < j < n$ tales que `abs(sum(l[i:j]))` sea máxima. Implementar el algoritmo en la función `max_abs_sum_sublst(l)` que devuelve el par `(i,j)`. ¿Cómo se podría maximizar `sum(l[i:j])`?

Sugerencia: Calcular la lista de las sumas parciales de l.