Universidad Nacional de la Patagonia San Juan Bosco

Laboratorio de Investigación en Informática

Conceptos Avanzados de Programación Python

Noviembre 2023

Parte 1

Índice

1. ¿Por qué necesitamos algoritmos?

Problema. Dado un número N se quiere calcular N^{33}. Una solución posible, por supuesto, es hacer el producto $N * N * ... * N$, que involucra 32 multiplicaciones.
Otra solución, más eficiente, es:

- Calcular $N * N$.
- Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de N^4.
- Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de N^8.
- Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de N^{16}.
- Al resultado anterior multiplicarlo por sí mismo con lo cual ya disponemos de N^{32}.
- Al resultado anterior multiplicarlo por N con lo cual conseguimos el resultado deseado con sólo 6 multiplicaciones.

Cada una de estas dos soluciones representa un algoritmo, es decir, un método de cálculo diferente. Para un mismo problema pueden haber algoritmos diferentes que lo resuelven, cada uno con un costo distinto en términos de recursos computacionales involucrados.

Hasta hace no mucho tiempo se utilizaba el término algoritmo para referirse únicamente a formas de realizar ciertos cálculos, pero con el surgimiento de la computación, el término algoritmo pasó a abarcar cualquier método para obtener un resultado.

El uso de la palabra *algorismo* proviene del nombre de un matemático persa famoso, en su época y para los estudiosos de esa época, Abu Abdallah Muhammad ibn Mûsâ al-Jwârizmî.

En la antigüedad, los *algoristas* eran los que calculaban usando la numeración arábiga y mientras que los *abacistas* eran los que calculaban usando ábacos. Con el tiempo el *algorismo* se deformó en *algoritmo*, influenciado por el término *aritmética*.

2. Cómo darle instrucciones a la máquina con Python

El lenguaje Python provee un *intérprete*, es decir un programa que interpreta las órdenes que le damos (en forma interactiva a medida que las escribimos). La forma más típica de invocar al intérprete es ejecutar el comando `python` en "modo consola" (usando *cmd*). Otra forma de ejecutarlo es a través de un script (programa guardado como un archivo de extensión .py) que contiene la secuencia de órdenes, escribiendo en la consola `python` seguido del nombre completo del archivo.

En https://www.python.org/downloads/ se encuentran los enlaces para descargar la última versión Python (y las anteriores).

En http://docs.python.org.ar/tutorial/3/interpreter.html hay más información acerca de cómo ejecutar el intérprete en cada sistema operativo.

```
>C: python
Python 3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15) [MSC
v.1929 64 bit (AMD64)] on win32
Type ``help``, ``copyright``, ``credits`` or ``license`` for more
information.
>>>
```

Para orientarnos, el intérprete muestra los símbolos ⋙ (llamaremos a esto el *prompt*), indicando que podemos escribir a continuación una *sentencia* u orden que será evaluada o interpretada.
Algunas sentencias sencillas, por ejemplo, permiten utilizar el intérprete como una calculadora simple con números (enteros o *floats* –reales). Para esto escribimos la *expresión* que queremos resolver luego del *prompt* y presionamos la tecla `Enter`. El intérprete de Python evalúa la expresión y muestra el resultado en la línea siguiente. Luego nos presenta nuevamente el *prompt*.

```
>>> 2+3
5
>>>
```

Python permite utilizar las operaciones +, -, *, / y ** (suma, resta, multiplicación, división y potenciación). La sintaxis es la convencional (valores intercalados con operaciones), y se puede usar paréntesis para modificar el orden de asociación natural de las operaciones (potenciación, producto/división, suma/resta). El práctico 1 que se presentará en la parte práctica de la materia introduce estos operadores y otros conceptos.

```
>>> 5*7
35
>>> 2+3*7
23
>>> (2+3)*7
35
>>> 10/4
2.5
>>> 5**2
25
```

Además de efectuar operaciones matemáticas, Python nos permite trabajar con porciones de texto, que llamaremos *cadenas* (que incluyen caracteres, dígitos, símbolos), y que se introducen entre comillas simples (') o dobles (”):

```
>>> '¡Hola Mundo!'
'¡Hola Mundo!'
>>> 'abcd' + 'efgh'
'abcdefgh'
>>> 'abcd' * 3
'abcdabcdabcd'
```

Python fue creado a finales de los años 80 por Guido van Rossum, quien sigue siendo aún hoy el líder del desarrollo del lenguaje. La versión 2.0, lanzada en 2000 era mucho más madura, incluyendo un recolector de basura. La última versión de esta línea es la 2.7 que fue lanzada en noviembre de 2010 y aún está vigente.

En diciembre de 2008 se lanzó la rama 3.0, cuya versión actual es la 3.10, de octubre de 2021, y es la que utilizamos en este curso. Python 3 fue diseñado para corregir algunos defectos de diseño en el lenguaje, y muchos de los cambios introducidos son incompatibles con las versiones anteriores. Por esta razón, las ramas 2.x y 3.x aún coexisten con distintos grados de adopción.

3. Variables

Python nos permite asignarle un nombre a un valor, de forma tal de "recordarlo" para usarlo posteriormente, mediante la sentencia `<nombre>=<expresión>`.

```
>>> x = 8
>>> x
8
>>> y = x * x
>>> 2 * y
128
>>> lenguaje = 'Python'
>>> 'Estoy programando en ' + lenguaje
'Estoy programando en Python'
```

En este ejemplo creamos tres variables, llamadas x, y y lenguaje, y las asociamos a los valores 8, 64 y 'Python', respectivamente. Luego podemos usar esas variables como parte de cualquier expresión, y en el momento de evaluarla, Python reemplazará las variables por su valor asociado.

4. Funciones

Para efectuar algunas operaciones particulares necesitamos introducir el concepto de *función*. Una función es un fragmento de programa que permite efectuar una operación determinada. `abs()`, `max()`, `min()` y `len()` y son ejemplos de funciones de Python.

La función `abs()` permite calcular el valor absoluto de un número, `max()` y `min()` permiten obtener el máximo y el mínimo entre un conjunto de números, y `len()` permite obtener la longitud de una cadena de texto.

> Funciones, programas, módulos, bibliotecas (libraries) son mecanismos para articular fragmentos de código con funcionalidades específicas, hechas para permitir el reuso, la portabilidad, la parametricidad, etc.

Una función puede recibir 0 o más *parámetros* o *argumentos* (expresados entre entre paréntesis, y separados por comas), efectúa una operación y devuelve un *resultado*. Por ejemplo, la función `abs()` recibe un parámetro (un número) y su resultado es el valor absoluto del número.

```
>>> abs(10)
10
>>> abs(-10)
10
>>> max(5, 9, -3)
9
>>> min(5, 9, -3)
-3
>>> len("abcd")
4
```

.

La mayoría de los programas que utilizamos cotidianamente no son más que grandes fragmentos de código implementados introduciendo nuevas funciones a la máquina, escritas por uno o muchos programadores.

Python viene equipado con muchas funciones. La mayoría de los programas que utilizamos cotidianamente no son más que grandes fragmentos de código implementados introduciendo nuevas funciones a la máquina, escritas por uno o muchos programadores. Nosotros luego también seremos creadores de funciones, programas, o sistemas completos.

5. Iteradores: el ciclo definido

Problema. Supongamos que queremos calcular la suma de los primeros 5 números cuadrados. Podemos hacer algo como esto:

Esto resuelve el problema, pero resulta poco útil. ¿Y si quisiéramos encontrar la suma de los primeros 100 números cuadrados? En ese caso tendríamos que repetir la línea `suma = suma + ... * ...` 100 veces. ¿Se puede hacer algo mejor que esto?

Para resolver este tipo de problema (repetir un cálculo para los valores contenidos en un intervalo dado) de una manera más eficiente, introducimos el concepto de *ciclo definido*, que tiene la siguiente forma:

```
>>> suma = 0
>>> suma = suma + 1*1
>>> suma = suma + 2*2
>>> suma = suma + 3*3
>>> suma = suma + 4*4
>>> suma = suma + 5*5
>>> suma
55
```

```
for x in range(n):
    <hacer algo con o sin x>
```

Esta instrucción se lee del siguiente modo:

- Generar la secuencia de valores enteros del intervalo [0, n), y
- Para cada uno de los valores enteros que toma x en el intervalo generado, se debe hacer lo indicado por `<hacer algo con x>`.

Notar que el intervalo es semiabierto. Se puede indicar también el valor inicial del intervalo:

```
for x in range(n1, n2):
    <hacer algo con x>
```

La instrucción que describe el rango en el que va a realizarse el ciclo `for x in range( )` es el *encabezado del ciclo*, y las instrucciones que describen la acción que se repite componen el *cuerpo del ciclo*.
Todas las instrucciones que describen el cuerpo del ciclo deben tener una *sangría* mayor que el encabezado del ciclo. Si hace falta más de una instrucción, se colocan todas con el mismo sangrado.

```
>>> suma = 0
>>> for x in range(6):
...     suma = suma + x*x
...
>>> suma
55
```

Notar el `...` luego de terminar el encabezado (lo escribe solo el intérprete), pero luego hay que dejar un sangrado al comenzar el cuerpo del ciclo. Notar también que al terminar el cuerpo del ciclo aparece otro `...` que pone el intérprete indicando que terminó de leer la línea. Se puede hacer todo en una sola línea, siempre que el cuerpo lo permita:

```
>>> suma = 0
>>> for x in range(6): suma = suma + x*x
>>> suma
55
```

El intérprete provee una ayuda en línea, nos puede dar la documentación de cualquier función o instrucción. Para obtenerla llamamos a la función `help()`. Si le pasamos por parámetro el nombre de una función (por ejemplo `help(abs)` o `help(range)`) nos dará la documentación de esa función. Para obtener la documentación de una instrucción la debemos poner entre comillas; por ejemplo: `help('for')`, `help('return')`.

6. Interacción con usuarias

Ya vimos que la función `print()` nos permite mostrar información a la usuaria del programa. En algunos casos también necesitaremos que ella ingrese datos al programa. Por ejemplo en el práctico pedimos un *script* (programa interpretado) que pida el nombre, y luego la salude.
La función `input()` interpreta cualquier valor que la usuaria ingresa mediante el teclado como una cadena de caracteres. Es decir, `input()` siempre devuelve una cadena, incluso aunque la usuaria haya ingresado una secuencia de dígitos.

```
$ python saludar.py
Por favor ingrese su nombre: Elisa
Hola Elisa!
```

El programa .py que tendremos como archivo será algo como lo siguiente:

```
nombre = input("Por favor ingrese su nombre: ")
saludo = 'Hola ' + nombre + '!'
print(saludo)
```

7. Tipos simples y operaciones en tipos simples

Los tipos de datos son conjuntos que determinan los valores admisibles que una variable o constante del tipo puede asumir.
Estos valores, y estos conjuntos, tienen una serie de características y propiedades determinadas. En Python, **todo valor que pueda ser asignado a una variable tiene asociado un tipo de dato, y por lo tanto se determina qué operaciones se pueden realizar sobre la misma**.

Los tipos de datos básicos de Python son los *booleanos*, los *numéricos* (enteros, punto flotante y complejos) y las *cadenas de caracteres*, que ya hemos visto, y que en principio son intuitivos de interpretar.
Python también define otros tipos de datos, entre los que se encuentran:

- Secuencias: Lista, tupla y rango
- Diccionarios
- Conjuntos
- Iteradores
- Clases
- etc.

8. Tipos numéricos

Python define tres tipos de datos numéricos básicos: *enteros*, *números de punto flotante* (simularía el conjunto de los números reales, pero ya veremos que no es así del todo) y los *números complejos*.

8.1. Números enteros

El tipo de los números *enteros* es `int`. Este tipo de dato comprende el conjunto de todos los números enteros, pero como dicho conjunto es infinito, en Python el conjunto está limitado realmente por la capacidad de la memoria disponible. No hay un límite de representación impuesto por el lenguaje. Un número de tipo int se crea a partir de una constante literal que represente un número entero o bien como resultado de una expresión o una llamada a una función. También podemos representar los números enteros en formato *binario*, *octal* o *hexadecimal* prefijando con 0b, 0o o 0x la constante.

```
>>> a = -1    # a es de tipo int y su valor es -1
>>> b = a + 2 # b es de tipo int y su valor es 1
>>> b
1
```

8.2. Números de punto flotante

Los números de *punto flotante* representan *aproximadamente* al conjunto de los números reales. ¿Qué entendemos por aproximadamente? No nos referimos a que solamente podemos representar *algunos* valores del conjunto, sino que tampoco podemos representar con exactitud los valores en sí (salvo excepciones).

La representación en cualquier base numérica de un número irracional es infinita, lo cual con la computadora no puede hacerse. La representación interna en la computadora (hecha con *bits*, o sea, números binarios) establece una relación de compromiso entre la *precisión* de la representación y la memoria asignada. Esta circunstancia tiene efectos a veces insospechados. En mi consola, por ejemplo, veo lo siguiente!!!:

```
>>> 1.1 + 2.2
3.3000000000000003      # whaaaaaaat???
```

8.3. Representación de los números de punto flotante

Tenemos que meternos en un detalle molesto para entender lo que ocurre (y otras cosas más). Para **representar** el mayor número posible de **reales** con las limitaciones de memoria (tamaños de palabra de una cantidad "práctica" de bits), se adaptó la **notación científica** de representación de números reales al sistema binario (que es el sistema que se utiliza en programación para representar los datos e instrucciones).
En esta notación, llamada de *mantisa* y *exponente*, los números se representan así:

Número	Notación científica
101,1	$1{,}011 * 10^2$
0,032	$3{,}2 * 10^-2$

Esto permite "negociar" cuántos bits se utilizan para los dígitos significativos, y cuántos para la cantidad de ceros a izquierda o a derecha del punto decimal (por eso se la denomina *punto flotante*). Si por alguna razón necesitásemos reales con mucha mayor precisión, existe el tipo de datos *decimal*.
El "problema" (ponele) es que la suma de la representación en punto flotante en binario del número 1,1 y de la representación en punto flotante en binario del número 2,2, da como resultado 3,3000000000000003. Pero hay más casos, como por ejemplo la representación del número 1/3. En algún momento, el ordenador tiene que truncar el número periódico resultante.
Más en detalle, la mantisa se representa como un cociente entre dos números binarios (enteros), lo cual da bastante amplitud (podemos representar a todos los números racionales dependiendo de la memoria que comprometamos), y siempre hay un racional "bastante cerca" de cualquier número real.

```
>>> Mi_real = 1.1 + 2.2 # Mi_real es un float
>>> Mi_real
3.3000000000000003
>>> print(f'{Mi_real:.2f}') # mostrando 2 cifras decimales
3.30
```

8.4. Números complejos

El último tipo de dato numérico básico que tiene Python es el de los números *complejos*, los `complex`. Los números complejos tienen una parte *real* y otra *imaginaria* y cada una de ellas se representa como un `float`. Para crear un número complejo, se sigue la siguiente estructura: `<parte_real> + <parte_imaginaria>j`. Se puede acceder a la parte real e imaginaria a través de los atributos real e imag.

```
>>> Mi_complejo = 1 + 2j #los espacios son supérfluos
>>> Mi_complejo.real
1.0
>>> Mi_complejo.imag
2.0
```

9. Aritmética de los tipos numéricos

Con todos los tipos numéricos se pueden aplicar las operaciones usuales de la aritmética: suma, resta, producto, división, exponentes, etc. algunos de los cuales ya vimos sin presentarlos formalmente (y en el práctico 1 se presentan ordenados por "prioridad" o precedencia).
En Python está permitido realizar una operación aritmética con números de distinto tipo. En este caso, el tipo numérico «menor» se convierte automáticamente al del tipo «mayor», de manera que el tipo del resultado siempre es de este último tipo (con `int <float <complex`). Estas "conversiones automáticas de tipos" son una fuente de problemas si no las conceptualizamos con cuidado. Por tanto, es posible, por ejemplo, sumar un `int` y un `float`, etc.

```
>>> entero = 2
>>> real = 2.1
>>> comple = 1 + 1j
>>> entero + real
4.1
>>> real + comple
(3.1+1j)
```

10. Tipo booleano

En Python el tipo que representa valores de verdad es el `bool`. Solo tiene dos valores: `True` para representar verdadero y `False` para representar falso.

George Boole (1815-1864) desarrolló un sistema de reglas que le permitían expresar, manipular y simplificar mecánicamente problemas lógicos cuyos valores admiten dos estados (verdadero o falso). Dicho sistema es un *álgebra de operadores lógicos* utilizado como base para la computación contemporanea.

Las constantes o variables booleanas se pueden combinar entre sí formando *expresiones lógicas*. Una característica importante de los lenguajes de programación es que permiten comparar valores escalares (reales, enteros, etc.) y determinar el valor de verdad de una comparación. A dichas expresiones se las denomina *expresiones de comparación*, y siempre evalúan a verdadero o falso. Las expresiones booleanas de comparación que provee Python son las siguientes:

Expresión	Significado
`a == b`	a es igual a b
`a != b`	a es distinto de b
`a < b`	a es menor que b
`a <= b`	a es menor o igual que b
`a > b`	a es mayor que b
`a >= b`	a es mayor o igual que b

En particular la pregunta de si x es mayor que cero, se codifica en Python como `x > 0`. De esta forma, `5 > 3` es una expresión booleana cuyo valor es `True`, y `5 < 3` también es una expresión booleana, pero su valor es `False`.

```
>>> 5 > 3
True
>>> 3 > 5
False
```

Algunos ejemplos (ver los *comparadores combinados* al final):

```
>>> 6 == 6
True
>>> 6 != 6
False
>>> 6 > 6
False
>>> 6 >= 6
True
>>> 6 > 4
True
>>> 6 <= 4
False
>>> 4 < 6
True
>>> real = 2.2
>>> entero = 3
>>> 2 < real < 3
True
2 < real < entero < 4
True
```

11. Operadores lógicos

De la misma manera que se puede operar entre números mediante las operaciones de suma, resta, etc., también existen tres operadores lógicos para combinar expresiones booleanas: `and` (y), `or` (o) y `not` (no). El significado de estos operadores es igual al del castellano, pero vale la pena recordarlo:

Expresión	Significado
`a and b`	El resultado es `True` solamente si a es `True` y b es `True` de lo contrario el resultado es `False`.
`a or b`	El resultado es `True` si a es `True` o b es `True` (o ambos) de lo contrario el resultado es `False`.
`not a`	El resultado es `True` si a es `False` de lo contrario el resultado es `False`.

Algunos ejemplos:

`a > b and a > c` es verdadero si a es simultáneamente mayor que b y que c.

```
>>> 5 > 2 and 5 > 3
True
>>> 5 > 2 and 5 > 6
False
```

`a > b or a > c` es verdadero si a es mayor que b o a es mayor que c.

```
>>> 5 > 2 or 5 > 3
True
>>> 5 > 2 or 5 > 6
True
>>> 5 > 8 or 5 > 6
False
```

`not a > b` es verdadero si `a > b` es falso (o sea si `a <= b` es verdadero).

```
>>> 5 > 8
False
>>> not 5 > 8
True
>>> 5 > 2
True
>>> not 5 > 2
False
```

Una particularidad (práctica pero a veces confusa) es que cualquier expresión puede ser usada en un contexto donde se requiera comprobar si algo es *verdadero* o *falso*. Por defecto, **cualquier valor (o expresión) es considerado como verdadero excepto**:

- `None`

- `False`
- El valor cero de cualquier tipo numérico: `0`, `0.0`, `0j`, ...
- Secuencias y colecciones vacías: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

12. Tipo cadena de caracteres

Otro tipo básico de Python es la secuencia o cadena de caracteres. Este tipo es conocido como *string* `str`. Formalmente, un string es una secuencia **inmutable** de caracteres en formato UNICODE.

> UNICODE es un estándar de codificación de caracteres diseñado para facilitar el tratamiento, transmisión y visualización de textos en numerosos idiomas. El término UNICODE proviene de sus tres características: universalidad, uniformidad y unicidad. UNICODE define cada carácter o símbolo mediante un nombre y un identificador numérico o código, y otras informaciones (sistema de escritura, categoría, direccionalidad, mayúsculas y otros atributos). UNICODE trata los caracteres alfabéticos, ideográficos y símbolos de forma igualitaria, se pueden mezclar en un mismo texto sin marcas o caracteres de control.

Para crear un string, rodeamos una secuencia de caracteres con comillas ‘ ’ o “ ”. Se puede usar indistintamente comillas simples o dobles, con una particularidad. Si el string contiene una comilla simple, se pueden usar comillas dobles para encerrar el string, o bien, usar comillas simples pero anteponer el carácter \ a la comilla simple del interior de la cadena. A diferencia de otros lenguajes, en Python no existe el tipo «carácter».

13. Tomando decisiones

Problema. Debemos leer un número entero y, si el número entero es positivo, debemos devolver “Número positivo”. Diseñamos nuestra solución:

1. Solicitar a la usuaria un número entero, guardarlo en x.
2. Si x >0, imprimir `"Número positivo"`

La primera línea requiere convertir la cadena de caracteres de la usuaria y convertirla a un entero (suponiendo que esto sea posible, caso contrario se generará un error).

```
x = int(input("Ingrese un número entero: "))
```

Para la segunda línea introducimos una nueva estructura de control que llamaremos *condicional* y tiene la siguiente forma:
y la probamos:

```
if <condición>:
    <hacer algo si se da la condición>
```

```
$ python positivo
Ingrese un número entero: 4
Número positivo
$ python positivo
Ingrese un número entero: -25
$ python positivo
Ingrese un número entero: 0
```

Problema. Necesitamos además un mensaje "Número no positivo" cuando no se cumple la condición.
Modificamos la especificación consistentemente y modificamos el diseño:

1. Solicitar a la usuaria un número entero, guardarlo en x.
2. Si x >0, imprimir "Número positivo"
3. Caso contrario, imprimir "Número no positivo"

La negación de x >0 se traduce en Python como `not x > 0` (o bien x >= 0), por lo que implementamos nuestra solución en Python como:

```
x = int(input("Ingrese un número entero: "))
if x > 0:
    print("Número positivo")
if not x > 0:
    print("Número no positivo")
```

Probamos la nueva solución y obtenemos el resultado buscado:

```
$ python positivo_o_no
Ingrese un número entero: 4
Número positivo
$ python positivo_o_no
Ingrese un número entero: -25
Número no positivo
$ python positivo_o_no
Ingrese un número entero: 0
Número no positivo
```

Sin embargo hay algo que nos preocupa: si ya averiguamos una vez si x >0, ¿Es realmente necesario preguntarlo de nuevo para negarlo ? Existe una construcción alternativa para la estructura de decisión: **Si se da la condición C, hacer S, de lo contrario, hacer T**. Esta estructura tiene la forma:
Nuestro script quedaría ahora

```
if <condición C>:
    <hacer tarea(s) S>
else:
    <hacer tarea(s) T>
```

```
x = int(input("Ingrese un número entero: "))
if x > 0:
    print("Número positivo")
else:
    print("Número no positivo")
```

14. Revisitando la asignación

Con sentencias sencillas de asignación como `x=4`, estamos estableciendo varios aspectos subyacentes:
Se crea un *objeto* (en este caso el `int 4`), se crea un identificador (en este caso `x`) y se crea una referencia, de manera que el identificador está *ligado* al objeto.
Los identificadores no tienen un tipo (los objetos si, y Python automáticamente lo determina), por lo que el tipo del identificador depende del objeto al cual haga referencia.

Python permite asignaciones múltiples en una misma línea (pronto veremos por qué):

```
x, y = 3, 4
```

Cuando el objeto es *inmutable*, su valor no cambia, y por lo tanto si se modifica el valor referenciado por un identificador, subyacentemente se crea **otro** objeto con el nuevo valor. Por ejemplo

```
x = 3 # se crea un nombre x, se crea un objeto 3, se crea una referencia
      #de x a ese objeto
x = x + 1 # se crea un objeto 4, x referencia a ese objeto, el objeto 3
          #queda "aislado"
```

Qué ocurre cuando una variable se asigna a otra variable?
`y = x`
En este caso, si `x` referencia a un objeto **inmutable**, podemos pensar que se crea otra instancia idéntica del mismo objeto referenciado por `x`, a la cual `y` ahora referenciará, o bien podemos pensar que `x` e `y` referencian al mismo objeto, pero que si luego `x` o `y` cambiasen, lo harían creando un nuevo objeto como se explicó arriba. Con los objetos mutables (por ejemplo un conjunto) este mecanismo será diferente, por lo que es indispensable conocer si un tipo es mutable o no para administrar correctamente las asignaciones.

15. Operaciones y conversiones de valores entre tipos

Operación	Operador	Comentario
Indexación	`s[i]`	Selecciona el i-ésimo caracter del string s, en base cero (`s[0]` es el primer caracter). Un índice negativo se cuenta desde el último elemento.
Rango	`s[i:j]`	Selecciona el rango de caracteres entre `[i:j)`.
Longitud	`len(s)`	Devuelve la cantidad de caracteres en `s`.
ASCII	`ord(c)`	Devuelve el valor ASCII de un caracter.
Caracter	`chr(x)`	Devuelve el "caracter" que corresponde a un código ASCII dado.
Conversión a string	`str(x)`	Convierte un valor escalar al string correspondiente.
Conversión a entero	`int(s)`	Convierte un string con una cadena de dígitos al valor entero correspondiente.
Conversión a real	`float(s)`	Convierte un string con un real al valor real correspondiente.

Que un string sea inmutable significa (entre otras cosas) que no puedo asignar a ninguno de sus caracteres

```
>>> str = 'abc'
>>> str[0] = 'x' # esto genera un error!!
```

ASCII es el acrónimo de American Standard Code for Information Interchange, y es un código de caracteres basado en el alfabeto latino, tal como se usa en inglés moderno. El código ASCII utiliza 7 bits para representar los caracteres. En la actualidad define códigos para 32 caracteres no imprimibles, de los cuales la mayoría son caracteres de control que tienen efecto sobre cómo se procesa el texto, más otros 95 caracteres imprimibles que les siguen en la numeración (empezando por el carácter espacio). Casi todos los sistemas informáticos actuales utilizan el código ASCII o una extensión compatible para representar textos y para el control de dispositivos que manejan texto como el teclado.

Algunos caracteres ASCII:

Un ejemplo: encontrar el mayor de tres enteros

16. Tuplas

La tupla es otros de los tipos inmutables de secuencia importantes en Python. se caracteriza por ser una secuenca de objetos entre ( ), separados por comas, pero los objetos

Binario	Dec	Hex	Representación	Binario	Dec	Hex	Representación	Binario	Dec	Hex	Representación
0010 0000	32	20	espacio ( )	0100 0000	64	40	@	0110 0000	96	60	`
0010 0001	33	21	!	0100 0001	65	41	A	0110 0001	97	61	a
0010 0010	34	22	"	0100 0010	66	42	B	0110 0010	98	62	b
0010 0011	35	23	#	0100 0011	67	43	C	0110 0011	99	63	c
0010 0100	36	24	$	0100 0100	68	44	D	0110 0100	100	64	d
0010 0101	37	25	%	0100 0101	69	45	E	0110 0101	101	65	e
0010 0110	38	26	&	0100 0110	70	46	F	0110 0110	102	66	f
0010 0111	39	27	'	0100 0111	71	47	G	0110 0111	103	67	g
0010 1000	40	28	(	0100 1000	72	48	H	0110 1000	104	68	h
0010 1001	41	29	)	0100 1001	73	49	I	0110 1001	105	69	i
0010 1010	42	2A	*	0100 1010	74	4A	J	0110 1010	106	6A	j
0010 1011	43	2B	+	0100 1011	75	4B	K	0110 1011	107	6B	k
0010 1100	44	2C	,	0100 1100	76	4C	L	0110 1100	108	6C	l
0010 1101	45	2D	-	0100 1101	77	4D	M	0110 1101	109	6D	m
0010 1110	46	2E	.	0100 1110	78	4E	N	0110 1110	110	6E	n
0010 1111	47	2F	/	0100 1111	79	4F	O	0110 1111	111	6F	o
0011 0000	48	30	0	0101 0000	80	50	P	0111 0000	112	70	p

```
>>> x = int(input("Ingresar el primer entero: "))
>>> y = int(input("Ingresar el segundo entero: "))
>>> z = int(input("Ingresar el tercer entero: "))
>>>     if y > x:
>>>         if z > y:
>>>            print(z," es el mayor.")
>>>         else:
>>>             print(y," es el mayor.")
>>>     else:
>>>         if z > x:
>>>             print(z," es el mayor.")
>>>         else:
>>>             print(x," es el mayor.")
>>> print("Listo!")
```

pueden ser heterogéneos.

```
>>> tupla = (2, 'hola', 3.14, 2+3j, (2, 3))
```

Observación: todos los elementos en la tupla son (o referencian a) valores.

```
>>> tupla = (a, b)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> a=1
>>> b=2
>>> tupla = (a,b)
>>> tupla
(1, 2)
```

En las tuplas son aplicables los mismos operadores de indexación que en los strings: También son aplicables los operadores + y *

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[1]
'abc'
>>> t[-3]
4.56
>>> t[1:4]
('abc', 4.56, (2,3))
>>> t[1:-1]
('abc', 4.56, (2,3))
 >>> t[:2]
(23, 'abc')
>>> t[2:]
(4.56, (2,3), 'def')
```

```
>>> t+(1,2)
(23, 'abc', 4.56, (2, 3), 'def', 1, 2)
>>> t*3
(23, 'abc', 4.56, (2, 3), 'def', 23, 'abc', 4.56, (2, 3), 'def', 23,
'abc', 4.56, (2, 3), 'def')
```

El operador `in` evalúa a un valor Booleano, tanto si un elemento está en una tupla como si si una sub-cadena está en un string. También se puede utilizar como **iterador** en un ciclo definido sobre los elementos de una estructura secuencial (cadena, tupla).

```
>>> t  # repetimos nuestro objeto para recordarlo
(23, 'abc', 4.56, (2, 3), 'def')
>>> 23 in t
True
>>> 'abc' in t
True
>>> 'bc' in t[1]
True
>>> 2 in t[3]
True
>>> 'de' in t
False
>>> 2 in t
False
>>> for x in (1, 'a', (1, 2)):
...     print(x)
...
1
a
(1, 2)
>>> string='abc'
>>> for x in string: print(x)
...
a
b
c
```

17. Iteración revisitada: ciclo indefinido

En muchas ocasiones necesitamos iterar una tarea, pero la cantidad necesaria de repeticiones es desconocida, dependiendo de que se cumpla (o no) una condición y no de la cantidad de repeticiones. Para estos casos necesitamos una estructura de control iterativa que repita la ejecución del cuerpo, pero dependiendo del cumplimiento de la mencionada condición. Como la cantidad de pasos no se especifica, la denominamos *iteración indefinida*, en la cual se repite la ejecución del cuerpo *mientras* una cierta condición es verdadera:

```
while <condición>:
  <ejecutar una tarea>
```

`while` es una palabra reservada, `condición` es una expresión booleana, igual que en las instrucciones condicionales (`if`). El cuerpo es, como siempre, una o más instrucciones del lenguaje. El funcionamiento de la instrucción while es el siguiente:

1. Evaluar la condición.
2. Si la condición evalúa a alguno de los valores nulos, salir del ciclo.
3. Caso contrario, ejecutar el cuerpo y volver a 1.

Recordamos los **valores nulos** en una expresión
`None`
`False`
El valor **cero** de cualquier tipo numérico: `0`, `0.0`, `0j`, `...`
Secuencias y colecciones **vacías**: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

¿Qué situación "riesgosa" se observa en la secuencia del funcionamiento del `while`?

Ejemplo: mi primer "sistema"
Habíamos mencionado que un sistema, a diferencia de un algoritmo, ejecuta su tarea de manera indefinida. Podemos tomar nuestro ejemplo de decidir si un número entero ingresado es positivo o no y transformarlo en un programa de ejecución contínua, colocándolo dentro de un ciclo while "infinito":

```
while True:
    x = int(input("Ingresá un número entero: "))
    if x > 0:
        print("Número positivo")
    elif x==0:
        print("Igual a cero")
    else:
        print("Número no negativo")
```

Claramente esto tiene algunas cosas para mejorar, por ejemplo poder salir del ciclo sin romper el intérprete:

```
HayMasDatos = "Si"
while HayMasDatos == "Si":
    x = int(input("Ingresá un número entero: "))
    if x > 0:
        print("Número positivo")
    elif x==0:
        print("Igual a cero")
    else:
        print("Número no negativo")
    HayMasDatos = input("¿Querés seguir? <Si-No>: ")
```

18. Mastermind

Con lo que ya sabemos del lenguaje, y un poco de "pensamiento algorítmico" estamos en condiciones de proponer cosas más interesantes. Pensemos en el juego **"Mastermind"**: hay que adivinar un número de cuatro dígitos distintos.
Queremos un programa que haga lo siguiente:

1. Elija un código secreto (**codigo**) de cuatro dígitos diferentes.
2. Pide a la usuaria una **propuesta** que adivine el código.
3. Si la usuaria adivinó la felicita y le indica cuántos **intentos** necesitó.
4. Caso contrario:
    - Le indica cuántos dígitos hay correctamente elegidos (coincidencias),
    - Le indica cuántos están correctamente colocados (aciertos),
    - Vuelve al paso 2.

Vamos a desensamblar esta especificación en partecitas, para que luego nos sirva como ejemplo de construcción "más racional" de programas complejos.

¿Cómo armar un string con cuatro dígitos diferentes y "aleatorios"? Comenzar con un string vacío. Repetir 4 veces lo siguiente:

1. Elegir un dígito al azar.
2. Mientras el dígito esté en el string, elegir otro.
3. Concatenar el dígito al string.

```
import random #necesitamos para elegir al azar

# para elegir el codigo
digitos = ('0','1','2','3','4','5','6','7','8','9')
codigo = ''
for i in range(4):
    candidato = random.choice(digitos)
    while candidato in codigo: candidato = random.choice(digitos)
    codigo = codigo + candidato
```

Bibliotecas (libraries) y funciones de biblioteca

Si bien el intérprete está en condiciones de reconocer y ejecutar una gran cantidad de funciones (por ejemplo `print`, `int`, `chr`, `etc`.) hay una gran cantidad (pero MUY grande!) de funciones disponibles. El intérprete no carga todas porque sería demasiado peso, entonces le delega a la programadora la decisión de incorporar en su script o programa las funciones que necesite, agrupadas "temáticamente" en bibliotecas (libraries).

Para avisarle a nuestro intérprete que utilizaremos una función de biblioteca, al comienzo del script utilizamos la declaración `import` seguido del nombre de la biblioteca (tenemos que saber cuál necesitamos, o buscarlo en Internet). Las declaraciones no son instrucciones o sentencias de nuestro programa, sino avisos al intérprete para que cargue la biblio correspondiente.

¿Cómo ver cuántos dígitos son `coincidencias` y cuántos `aciertos`?

1. Asignar `aciertos, coincidencias = 0, 0`
2. Para `i` entre `0` y `3` repetir:
   - Si `propuesta[i] == codigo[i]` aumentamos el valor de `aciertos`
   - Caso contrario si `propuesta[i] in codigo` aumentamos `coincidencias`
3. Sí `aciertos == 4` la usuaria ganó!

Queremos ahora que el juego "cicle" hasta que la jugadora gane, cuente cuántas veces la jugadora intentó, y que cuando gane le indique la cantidad de `intentos`:

```
# para determinar si ganaste
aciertos = 0
coincidencias = 0
for i in range(4):
    if propuesta[i] == codigo[i]:
        aciertos = aciertos + 1
    elif propuesta[i] in codigo:
        coincidencias = coincidencias + 1
if aciertos == 4:
    ganaste = True
```

1. Saludar, inicializar `intentos` (a cero), inicializar `ganaste` (a falso)
2. Elegir el `codigo` secreto
3. Pedir la `propuesta` a la jugadora
4. Mientras `ganaste` sea falso repetir
    - Incrementar `intentos`
    - Contar `aciertos` y `coincidencias`
    - Si `aciertos == 4`
      felicitar a la jugadora
      indicar cuántos `intentos`
      asignar `ganaste = True`
    - Caso contrario
      avisar cuántos `aciertos` y `coincidencias`
      pedir nueva `propuesta`

```
print("Bienvenidx al Mastermind!") # etc etc
intentos = 0
ganaste = False
# acá viene elegir el código
propuesta = input("Que código proponés?: ")
while not ganaste:
    intentos = intentos + 1
# acá viene determinar aciertos y concidencias
    if aciertos == 4:
        ganaste = True
        print("Felicitaciones, has ganado ...) # etc. etc.
    else:
        print("Tu propuesta tiene tantos aciertos...) # etc. etc.
        propuesta = input("Elegí otro código: ")
```

El script final queda así:

```
import random
print("Bienvenidx al Mastermind!")
print("Tenés que adivinar un número de cuatro dígitos distintos")
intentos = 0
ganaste = False

# para elegir el código
digitos = ('0','1','2','3','4','5','6','7','8','9')
codigo = ''
for i in range(4):
    candidato = random.choice(digitos)
    while candidato in codigo:
        candidato = random.choice(digitos)
    codigo = codigo + candidato

propuesta = input("Que código proponés?: ")

while not ganaste:
    intentos = intentos + 1
    aciertos = 0
    coincidencias = 0
    for i in range(4):
        if propuesta[i] == codigo[i]:
            aciertos = aciertos + 1
        elif propuesta[i] in codigo:
            coincidencias = coincidencias + 1
    if aciertos == 4:
        ganaste = True
        print("Felicitaciones, has ganado en "+str(intentos)+" intentos!")
    else:
        print("Tu propuesta "+str(propuesta)+" tiene "+str(aciertos)+"
                aciertos y "+str(coincidencias)+" coincidencias")
        propuesta = input("Elegí otro código: ")
```

19. Definición de funciones

El programa queda un poco "largo" y requiere "subir y bajar" la lectura para entenderlo y no perderse. Esa es siempre una práctica dudosa y riesgosa. Para poder *factorizar* nuestro programa en unidades conceptualmente adecuadas, podemos utilizar la definición de funciones.

```
def saludar(usuarie):
    print("Hola " + usuarie + "!")
```

`def` es una palabra reservada que avisa al intérprete que lo que sigue (de acuerdo con la indentación y espacios) es la definición de una función.

Las funciones pueden recibir ninguno, uno, o varios parámetros, y devolver ninguno, uno, o varios resultados. Es importante observar que los parámetros de la función son evaluados cuando ésta se invoca.

```
#Se puede "comentar" la definición de una función para que luego la ayuda
#online nos recupere el comentario.

def saludar(usuarie):
    """Saluda a la persona indicada por parámetro."""
    print("Hola " + usuarie + "!")

>>> saludar('profe')
Hola profe!
>>> help(saludar)
Help on function saludar in module __main__:

saludar(usuarie)
    Saluda a la persona indicada por parámetro.
```

Volviendo al Mastermind, ¿cómo utilizar funciones para hacerlo más "articulado"?

```
def elegir_codigo(): # no recibe parámetros!!!
    """ Devuelve un código de 4 dígitos elegido al azar."""

digitos = ('0','1','2','3','4','5','6','7','8','9')
codigo = ''

for i in range(4):
    candidato = random.choice(digitos)
    while candidato in codigo:
        candidato = random.choice(digitos)
    codigo = codigo + candidato

return codigo
```

`return` es una palabra reservada que como imaginamos le indica al intérprete que la función terminó y debe devolver el valor indicado a continuación.

```
def analizar(propuesta, codigo):
    """ Devuelve la cantidad de aciertos y coincidencias."""

aciertos = 0
coincidencias = 0
```

```
for i in range(4):
    if propuesta[i] == codigo[i]:
        aciertos = aciertos + 1
    elif propuesta[i] in codigo:
        coincidencias = coincidencias + 1
return aciertos, coincidencias # devuelve una tupla!!
```

Estamos aprovechando la característica de Python de que las tuplas son tipos inmutables, y por lo tanto una función puede devolver (en una tupla) más de un valor. Esta característica es muy ventajosa.
Finalmente nuestro script queda (ponele) más elegante

```
import random

<--- aquí colocamos las definiciones de las funciones --->

print("Bienvenidx al Mastermind!")
print("Tenés que adivinar un número de cuatro dígitos distintos")
codigo = elegir_codigo()
propuesta = input("Que código proponés?: ")
intentos = 1
while propuesta != codigo:
    aciertos, coincidencias = analizar(propuesta, codigo)
    print("Tu propuesta "+str(propuesta)+" tiene "+str(aciertos)+
            "aciertos y "+str(coincidencias)+" coincidencias")
    propuesta = input("Elegí otro código: ")
    intentos = intentos + 1
print("Felicitaciones, has ganado en "+str(intentos)+" intentos!")
```

20. Alcance de las variables

Podemos declarar variables dentro del cuerpo de una función. Estas variables serán solamente visibles y accesibles *dentro* de la misma. Las variables y los *parámetros* que se declaran dentro de una función no existen fuera de ella, y por eso se las denomina *variables locales*. Fuera de la función se puede acceder únicamente al valor que devuelve la función mediante `return`.
Las variables declaradas dentro de una función pueden tener nombres iguales a otras variables (existentes fuera de la función, o existentes dentro de otras funciones) pero referencian a objetos diferentes y por lo tanto "no se mezclan".

```
def funcion_1():
    x = 'este es el x de la funcion 1'
    funcion_2()
    print(x)

def funcion_2():
    x = 'este es el x de la funcion 2'
    print(x)

x = 'este es el x del programa principal'
funcion_1()
funcion_2()
print(x)
```

El resultado de ejecutar este script es el siguiente:

```
este es el x de la funcion 2
este es el x de la funcion 1
este es el x de la funcion 2
este es el x del programa principal
```

Sin embargo, si dentro de una función referenciamos a una variable que no está declarada dentro de la misma, el intérprete buscará un identificador fuera de la función (hasta ahora, en el programa principal). Ver cómo se modifica el comportamiento en el siguiente ejemplo:

```
def funcion_1():
    #x = 'este es el x de la funcion 1'
    funcion_2()
    print(x)

def funcion_2():
    x = 'este es el x de la funcion 2'
    print(x)

x = 'este es el x del programa principal'
funcion_1()
funcion_2()
print(x)
```

El resultado de ejecutar el script anterior es el siguiente:

```
este es el x de la funcion 2
este es el x del programa principal
este es el x de la funcion 2
este es el x del programa principal
```

Es posible también declarar una función dentro de otra (lo cual hace que la primera sea visible sólo dentro de la segunda). Las reglas de alcance de las variables se aplican de idéntico modo.

21. Un ejemplo 'complejo': Bhaskara

Queremos recibir los tres coeficientes de una ecuación de segundo grado y encontrar sus raíces si es que éstas existen. La idea es definir una función con tres parámetros numéricos (reales o enteros) que devuelva dos string con las dos raíces, un solo string no vacío si es un sistema lineal, y dos strings vacíos si el sistema no tiene solución:

```
a = int(input("Ingrese el coeficiente de la variable cuadrática\n"))
b = int(input("Ingrese el coeficiente de la variable lineal\n"))
c = int(input("Ingrese el término independiente\n"))

e, f = bsk(a,b,c)
if e:
  if f:
    print('raiz 1 = ' + e)
    print('raiz 2 = ' + f)
  else:
    print("Sistema lineal")
    print('solución = ' + e)
else: print("No tiene solución")
```

Tenemos que considerar todos los casos posibles:

```
from math import sqrt

def bsk(x2, x1, x0):
if x2 == 0:
    if x1 == 0: return '', ''    # no tiene solución
    else: return str(-x0/x1), '' # sistema lineal
else:
    discr = x1**2 - 4*x2*x0
    if discr >= 0:
        r1 = (-x1 + sqrt(discr))/(2*x2)
        r2 = (-x1 - sqrt(discr))/(2*x2)
        return str(r1), str(r2)
    else:
        re = -x1/(2*x2)
        im = sqrt(-discr) / (2*x2)
        return str(complex(re,im)), str(complex(re,-im))
```

22. Revisitando nuevamente la asignación

Python permite asignaciones en tuplas (esto se denomina "empaquetado" y "desempaquetado").

```
>>> a = 125
>>> b = "#"
>>> c = "Ana"
>>> d = a, b, c
>>> d
(125, '#', 'Ana')
>>> x, y, z = d
>>> x
125
>>> y
'#'
>>> z
'Ana'
```

Hay casos de empaquetado y desempaquetado que requieren consideración:

```
>>> p, p, p = d
>>> p
'Ana'

>>> m, n = d
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: too many values to unpack

>>> m, n, o, p = d
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
```

En la implementación de algunos programas suele ser necesario intercambiar el valor de dos variables. Si queremos intercambiar el valor de a y b, una posibilidad es utilizar una variable auxiliar:

```
aux = a
a = b
b = aux
```

Un truco que podemos aplicar es empaquetar y desempaquetar una tupla en una única operación:

```
a, b = b, a
```

23. Listas

La lista será nuestro primer tipo de dato mutable. Es una secuencia ordenada de elementos (como es la tupla) pero en la lista se pueden modificar los elementos, agregar, quitar, etc. A diferencia de la tupla, se demarca con `'['` y `']'`.

```
lista_de_enteros = [1, 3, 900]
lista_heterogenea = ["Hola", 1, 2.34]
lista_con_listas = ["Elemento', [1, 2, 3], [3.4, "casa']]
lista_vacia = []
```

Para acceder a los elementos de una lista utilizamos la misma notación que para strings:

```
>>> lista_enteros[2]
900
>>> lista_heterogenea[0]
'Hola'
>>> lista_con_listas[1]
[1, 2, 3]
>>> lista_con_listas[2]
[3.4, 'casa']
>>> lista_con_listas[2][1]
'casa'
```

Muchos procesos se realizan con iteradores sencillos:

```
Cant = 0
for elemento in lista_enteros :
    Cant = Cant + 1
print('La cantidad de elementos en la lista es: ', cant)

Sumatoria = 0
for elemento in lista_enteros :
    Sumatoria = Sumatoria + elemento
print('La suma de los elementos en la lista es: ', Sumatoria)

Mayor = lista_enteros[0]
for elemento in lista_enteros:
    if elemento > Mayor :
        Mayor = elemento
print("El mayor elemento en la lista es: ", Mayor )
```

Ejemplo un poco más elaborado: buscar un elemento y determinar en qué posición se encuentra, o bien si no está:

```
Buscado = int(input("Ingrese el elemento que desea buscar"))

# Inicializamos los valores relevantes
Pos = 0
Lo_Encontre = False

# Recorremos la lista hasta encontrarlo o hasta que se termine
while Pos < len(lista_enteros) and not Lo_Encontre :
    if lista_enteros[Pos] == Buscado :
        Lo_Encontre = True
    else:
        Pos = Pos + 1
# salimos del ciclo: o lo encontré o ya revisé toda la lista
if Lo_Encontre :
    print("Encontré el elemento en la pos", Pos )
else:
    print("El elemento no está en la lista.")
```

Los operadores usuales en otras colecciones también funcionan con listas.

```
>>> L = [1, 2, 3]
>>> L
[1, 2, 3]
>>> L2 = L + L
>>> L2
[1, 2, 3, 1, 2, 3]
>>> L3 = L*3
>>> L3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> LL = [L, L]
>>> LL
[[1, 2, 3], [1, 2, 3]]
>>> L3[1:4]
[2, 3, 1]
>>> LL[1][1]
2
```

Lo interesante con las listas es que podemos modificarlas:

```
>>> for i in range(len(L)) : L[i] = 0.5 + L[i]
...
>>> L
[1.5, 2.5, 3.5]
>>> L[1:3] = 4, 5, 6
>>> L
[1.5, 4, 5, 6]
```

Y también transformarlas

```
>>> LT = [elemento * 3 for elemento in L]
>>> LT
[4.5, 12, 15, 18]
>>> LE = [elemento for elemento in L if elemento > 4]
>>> LE
[5, 6]
```

Algunos de los métodos sobre listas:

```
>>> lista = [1,2,3,4,5]
>>> lista.append(6)
[1, 2, 3, 4, 5, 6]
>>> lista.insert(1, 1.5) # no confundir con lista[1] = 1.5
[1, 1.5, 2, 3, 4, 5, 6]
>>> lista.index(5)
5
>>> lista.pop(0)
1
>>> lista
[1.5, 2, 3, 4, 5, 6]
>>> lista.pop(2)
3
>>> lista
[1.5, 2, 4, 5, 6]
```

Triángulo de Pascal

Es una estructura matemática que se obtiene en cada celda al sumar los dos números que están inmediatamente arriba (al "NE" y al "NO"), considerando en los extremos que el exterior del triángulo contiene todos ceros. De esa forma se obtienen todos los coeficientes binomiales y números combinatorios. Cada fila representa el orden del binomio (o la cantidad de elementos) y la "columna" el término del binomio o la cantidad de elementos a extraer.

1	$\binom{0}{0}$
1 1	$\binom{1}{0}$ $\binom{1}{1}$
1 2 1	$\binom{2}{0}$ $\binom{2}{1}$ $\binom{2}{2}$
1 3 3 1	$\binom{3}{0}$ $\binom{3}{1}$ $\binom{3}{2}$ $\binom{3}{3}$
1 4 6 4 1	↑
1 5 10 10 5 1	Última fila correspondiente a $(a+b)^3$

Quisiéramos (primero) armar un programa que dado N escriba las primeras N filas del triángulo. Para ello definimos una función que recibe una fila (representada como una lista de enteros) y devuelve la fila inferior de acuerdo a la definición. Esta función primero coloca ceros al comienzo y fin de la fila recibida, y luego genera una fila nueva sumando los elementos correlativos de a dos:

```
def nueva_fila(fila):
    fila.insert(0,0)
    fila.append(0)
    nf = []

    for i in range(len(fila)-1) :
        nf.append(fila[i] + fila[i+1])

    return(nf)
```

Luego, definimos una función que a partir de la fila 0 (que sería la lista [1]) genere la fila uno, luego la dos, etc., hasta llegar al N provisto por la usuaria:

```
def pascal(N):
    fila = [1]
    for i in range(N) :
        print(fila)
        fila = nueva_fila(fila)
```

Observar que con N=0 el ciclo for no se ejecuta, con N=1 se ejecuta una sola vez e imprime la primera fila (que es la fila 0), etc. Finalmente, el programa solo pide el valor de N y llama a esta última función.

```
Ingrese la cantidad de filas deseada 8
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

Más métodos sobre listas

Para eliminar un elemento de una lista se utiliza el método `remove`. Aquí no nos interesa el *lugar* en el que está el elemento, sino su *valor*. Si hay más de una aparición se elimina la primera. Si el elemento no existe se genera un error.

```
>>> L = [1, 2, 3, 4, 5, 1]
>>> L.remove(1)
>>> L
[2, 3, 4, 5, 1]
>>> L.remove(1)
>>> L
[2, 3, 4, 5]
>>> L.remove(1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Listas, tuplas y strings son tipos de valor secuencia. Las funciones `list` y `tuple` generan listas o tuplas a partir de otras estructuras de secuencia, y por eso se las llama *funciones constructoras*.

```
>>> list('hola')
['h', 'o', 'l', 'a']
>>> list((1, 2))
[1, 2]
>>> list(['a', 'b'])
['a', 'b']

>>> tuple('hola')
('h', 'o', 'l', 'a')
>>> tuple([1, 2])
(1, 2)
>>> tuple(('a', 'b'))
('a', 'b')
```

El método `split()` aplicado a una cadena de caracteres permite armar una lista respecto de una subcadena separadora.

```
>>> string = "Uno, dos, tres, cuatro"
>>> string.split(' ')
['Uno,', 'dos,', 'tres,', 'cuatro']
>>> string.split(',')
['Uno', ' dos', ' tres', ' cuatro']
>>> string.split(', ')
['Uno', 'dos', 'tres', 'cuatro']
>>> string.split('res')
['Uno, dos, t', ', cuatro']
```

A la inversa, el método `join()` aplicado a una cadena separadora, recibe una lista de elementos y los une en un string separándolos por dicha cadena.

```
>>> "@".join(['a', 'b'])
'a@b'
```

```
>>> "-".join(list("hola"))
'h-o-l-a'

>>> " + ".join(list("hola"))
'h + o + l + a'
```

La función `sorted()` recibe una lista y devuelve otra lista ordenada.

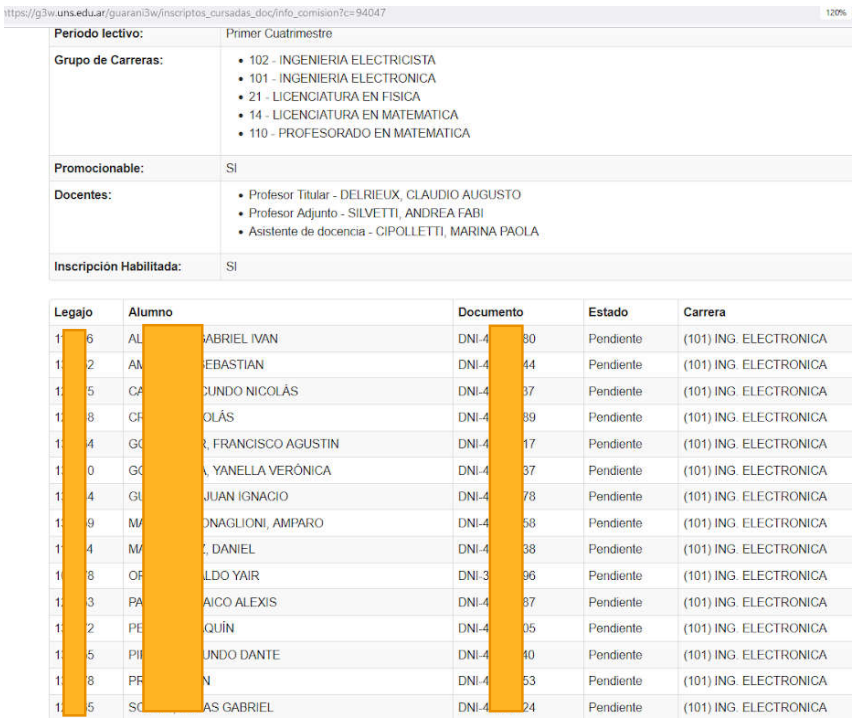
```
>>> bs = [5, 2, 4, 2]
>>> cs = sorted(bs)
>>> bs
[5, 2, 4, 2]
>>> cs
[2, 2, 4, 5]
```

El método `sort()` aplicado a una lista, ordena *la misma lista*.

```
>>> ds = [5, 3, 4, 5]
>>> ds.sort()
>>> ds
[3, 4, 5, 5]
```

Un pequeño sistema de información

Supongamos que queremos hacer un pequeño sistema de información, por ejemplo para el manejo de legajos. Tomamos los datos del SIU-Guaraní utilizando un web-scrapper (veremos este tema más adelante).



https://g3w.uns.edu.ar/guarani3w/inscriptos_cursadas_doc/info_comision?c=94047 120%

Periodo lectivo:	Primer Cuatrimestre
Grupo de Carreras:	• 102 - INGENIERIA ELECTRICISTA • 101 - INGENIERIA ELECTRONICA • 21 - LICENCIATURA EN FISICA • 14 - LICENCIATURA EN MATEMATICA • 110 - PROFESORADO EN MATEMATICA
Promocionable:	SI
Docentes:	• Profesor Titular - DELRIEUX, CLAUDIO AUGUSTO • Profesor Adjunto - SILVETTI, ANDREA FABI • Asistente de docencia - CIPOLLETTI, MARINA PAOLA
Inscripción Habilitada:	SI

Legajo	Alumno	Documento	Estado	Carrera
1[redacted]6	AL[redacted]GABRIEL IVAN	DNI-4[redacted]80	Pendiente	(101) ING. ELECTRONICA
1[redacted]2	AM[redacted]EBASTIAN	DNI-4[redacted]44	Pendiente	(101) ING. ELECTRONICA
1[redacted]5	CA[redacted]CUNDO NICOLÁS	DNI-4[redacted]37	Pendiente	(101) ING. ELECTRONICA
1[redacted]8	CR[redacted]OLÁS	DNI-4[redacted]89	Pendiente	(101) ING. ELECTRONICA
1[redacted]4	GO[redacted]R, FRANCISCO AGUSTIN	DNI-4[redacted]17	Pendiente	(101) ING. ELECTRONICA
1[redacted]0	GO[redacted]A, YANELLA VERÓNICA	DNI-4[redacted]37	Pendiente	(101) ING. ELECTRONICA
1[redacted]4	GU[redacted]JUAN IGNACIO	DNI-4[redacted]78	Pendiente	(101) ING. ELECTRONICA
1[redacted]9	MA[redacted]ONAGLIONI, AMPARO	DNI-4[redacted]58	Pendiente	(101) ING. ELECTRONICA
1[redacted]4	MA[redacted]Z, DANIEL	DNI-4[redacted]38	Pendiente	(101) ING. ELECTRONICA
1[redacted]8	OR[redacted]LDO YAIR	DNI-3[redacted]96	Pendiente	(101) ING. ELECTRONICA
1[redacted]3	PA[redacted]AICO ALEXIS	DNI-4[redacted]87	Pendiente	(101) ING. ELECTRONICA
1[redacted]2	PE[redacted]QUÍN	DNI-4[redacted]05	Pendiente	(101) ING. ELECTRONICA
1[redacted]5	PI[redacted]UNDO DANTE	DNI-4[redacted]40	Pendiente	(101) ING. ELECTRONICA
1[redacted]8	PR[redacted]N	DNI-4[redacted]53	Pendiente	(101) ING. ELECTRONICA
1[redacted]5	SC[redacted]AS GABRIEL	DNI-4[redacted]24	Pendiente	(101) ING. ELECTRONICA

Esa página web (en HTML) transforma (después de seleccionar las tres primeras columnas y filtrar caracteres innecesarios, y aplicarle un anonimizador) en la siguiente lista:

```
[['241251', 'Adobatrk, Gobtrya Itun', '49114401']
['259497', 'Anotrosso, Sobasskyan', '51953065']
['249120', 'Cassktro, Fahwndo Nyhoáss', '49768958']
['249593', 'Ctryhho, Nyhoáss', '50306810']
['253509', 'Gogg Aatr, Ftranhyssho Agwsskyn', '50695238']
['260755', 'Gonaz Latra, Yania Vatrónyha', '51975758']
['255089', 'Gwasratro, Jwan Ignahyo', '51626399']
['259304', 'Matrkínaz Sonagyony, Anpatro', '52144779']
['235509', 'Mayo Patraz, Danya', '47955159']
['230823', 'Otrkyz, Ronado Yaytr', '45747617']
['253198', 'Pahhaho, Mayho Aaxyss', '50504008']
['259617', 'Paktrwf, Joaqwín', '51545526']
['257800', 'Pytroa, Fahwndo Danka', '52535461']
['259723', 'Ptrak, Fatrnín', '50959174']
['250110', 'Shhnyk, Lwhass Gobtrya', '49970445']]
```

Queremos hacer un buscador por clave, que nos permita ver si une alumne está en la lista, buscando el nombre, el DNI o el legajo a partir de alguno de los otros datos. Colocamos dicho *"data frame"* (en este caso una lista de listas) en una variable `Legajos`. Supongamos que queremos buscar el nombre, a partir del DNI o el legajo. Armamos una función que reciba DNI o legajo, busque en `Legajos`, y devuelva el nombre del registro (si lo encuentra):

```
def Buscar_Nombre(Buscado) :
    Dato = [e for e in Legajos if Buscado in e]
    if Dato : return(Dato[0][1])
    else : return("No se encontró la persona")
```

Es fácil ver los demás casos. Luego, necesitamos iterar el proceso de preguntar a la usuaria qué dato desea buscar. Lo podemos hacer de muchas maneras, por ejemplo pidiendo una letra.

```
    Flag = input("Ingrese qué desea buscar ('N' Nombre, 'L'
                  Legajo, 'D' DNI, '' para salir: ")
    while Flag :
        if Flag == "N" :
            Buscado = input("Ingrese DNI o Legajo: ")
            print(Buscar_Nombre(Buscado))
        elif Flag == "L" :
            Buscado = input("Ingrese DNI o Nombre: ")
            print(Buscar_Legajo(Buscado))
        elif Flag == "D" :
            Buscado = input("Ingrese Nombre o Legajo: ")
            print(Buscar_DNI(Buscado))
        Flag = input("Ingrese qué desea buscar ('N' Nombre, 'L'
                      Legajo, 'D' DNI, '' para salir: ")
    print("Gracias por usar nuestros servicios (ponele)")

 . . .

Ingrese qué desea buscar ('N' Nombre, 'L' Legajo, 'D' DNI, '' para salir: N
Ingrese DNI o Legajo: 241251
Adobatrk, Gobtrya Itun

Ingrese qué desea buscar ('N' Nombre, 'L' Legajo, 'D' DNI, '' para salir: D
Ingrese Nombre o Legajo: 241251
49114401

Ingrese qué desea buscar ('N' Nombre, 'L' Legajo, 'D' DNI, '' para salir: L
Ingrese DNI o Nombre: Adobatrk, Gobtrya Itun
241251

Ingrese qué desea buscar ('N' Nombre, 'L' Legajo, 'D' DNI, '' para salir:
Gracias por usar nuestros servicios (ponele!)
```

24. Revisitando otra vez más la asignación

La asignación en objetos mutables tiene un comportamiento diferente. Si asignamos un objeto mutable (por ahora una lista) a una nueva variable, esta variable tendrá como referencia al *mismo objeto mutable* (no se hace una copia de su valor).
A este mecanismo por el cual un mismo objeto puede tener dos (o más referencias) se lo denomina *"aliasing"*. Esta posibilidad puede parecer extraña pero tiene su razón de ser. En muchos problemas de programación que veremos, es necesaria la posibilidad de recibir por parámetro parte de una estructura para luego modificarla. Pensemos por ejemplo en un cursor, en un editor de texto. El archivo está siendo referenciado por una variable (el nombre del archivo), y el cursor al comienzo referencia a la misma estructura, pero luego el cursor puede "navegar" dentro de la misma. Si luego alteramos lo referenciado por el cursor, queremos que el archivo registre dicho cambio.

```
>>> L = [1,2,3,4]
>>> L
[1, 2, 3, 4]
>>> L1=L
>>> L1
[1, 2, 3, 4]
>>> L.pop()
4
>>> L1
[1, 2, 3]
>>> L
[1, 2, 3]
```

Cuando al alterar el valor de una referencia se altera el valor de alguna otra, a eso se lo denomina ”efecto colateral“.

Supongamos ahora que quisiéramos una salida de la tabla ordenada. Haciendo `for e in sorted(Legajos) : print(e)` obtenemos una salida ordenada por el primer campo de cada registro:

```
['230823', 'Otrkyz, Ronado Yaytr', '45747617']
['235509', 'Mayo Patraz, Danya', '47955159']
['241251', 'Adobatrk, Gobtrya Itun', '49114401']
['249120', 'Cassktro, Fahwndo Nyhoáss', '49768958']
['249593', 'Ctryhho, Nyhoáss', '50306810']
['250110', 'Shhnyk, Lwhass Gobtrya', '49970445']
['253198', 'Pahhaho, Mayho Aaxyss', '50504008']
['253509', 'Gogg Aatr, Ftranhyssho Agwsskyn', '50695238']
['255089', 'Gwasratro, Jwan Ignahyo', '51626399']
['257800', 'Pytroa, Fahwndo Danka', '52535461']
['259304', 'Matrkínaz Sonagyony, Anpatro', '52144779']
['259497', 'Anotrosso, Sobasskyan', '51953065']
['259617', 'Paktrwf, Joaqwín', '51545526']
['259723', 'Ptrak, Fatrnín', '50959174']
['260755', 'Gonaz Latra, Yania Vatrónyha', '51975758']
```

Podríamos pensar ahora para tener una lista ordenada por nombre, “rotar” cada registro para que quede el nombre al frente, y luego ordenar la lista con los registros rotados:

```
L2 = Legajos
for e in L2 : e.append(e.pop(0))
L2.sort()
for e in L2 : print(e)
```

Esto va a dar resultado, pero en realidad `L2` es un alias de `Legajos`! Para evitar alterar a esta última, podríamos utilizar el método copy en la primera línea:

```
L2 = Legajos.copy()
```

Esto da resultado "superficialmente" (se crea una nueva lista, pero los elementos de la misma son "alias" de los elementos de la lista anterior).

Podríamos pensar entonces en copiar una lista a la cual le copiamos los elementos (y así recursivamente). Por suerte esto ya está implementado en el método `deepcopy()` de la clase copy.

```
import copy
...
L2 = copy.deepcopy(Legajos)
...
```

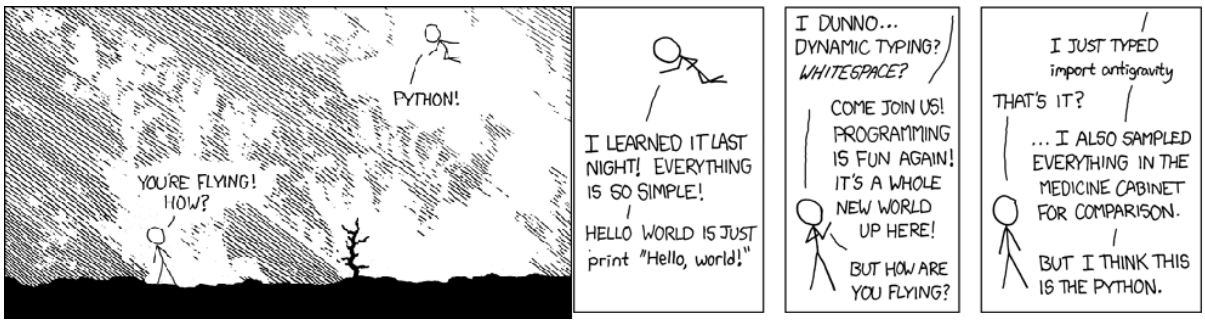
Con esto obtenemos el resultado buscado sin alterar Legajos:

```
['Adobatrk, Gobtrya Itun', '49114401', '241251']
['Anotrosso, Sobasskyan', '51953065', '259497']
['Cassktro, Fahwndo Nyho´ass', '49768958', '249120']
['Ctryhho, Nyho´ass', '50306810', '249593']
['Gogg Aatr, Ftranhyssho Agwsskyn', '50695238', '253509']
...
['Pahhaho, Mayho Aaxyss', '50504008', '253198']
['Paktrwf, Joaqw´ın', '51545526', '259617']
['Ptrak, Fatrn´ın', '50959174', '259723']
['Pytroa, Fahwndo Danka', '52535461', '257800']
['Shhnyk, Lwhass Gobtrya', '49970445', '250110']
```

Finalmente, para ordenar por DNI hacemos la rotación opuesta:

```
L3 = copy.deepcopy(Legajos)
for e in L3 : e.insert(0,(e.pop()))
L3.sort()
for e in L3 : print(e)
```

Es entonces importante ver que para cualquier problema "razonable" que tengamos, sepamos buscar el `import` que nos hace falta.



25. Repaso de lo visto hasta ahora

- `(valor1, valor2, valor3)`

  Las tuplas se definen como una sucesión de valores encerrados entre paréntesis y separados por comas. Una vez definidas, no se pueden modificar los valores asignados. Casos particulares:

```
tupla_vacia = ()
tupla_unitaria = (3459,)
```

- `[valor1, valor2, valor3]`

  Las listas se definen como una sucesión de valores encerrados entre corchetes y separados por comas. Se les puede agregar, quitar o cambiar los valores que contienen. Caso particular:

```
lista_vacia = []
```

- `x, y, z = secuencia`

  Es posible desempaquetar una secuencia, asignando a la izquierda tantas variables como elemen- tos tenga la secuencia. Cada variable tomará el valor del elemento que se encuentra en la misma posición.

- `len(secuencia)`

  Devuelve la cantidad de elementos que contiene la secuencia, 0 si está vacía.

- `for elemento in secuencia:`

  Itera uno a uno por los elementos de la secuencia.

- `elemento in secuencia`

  Indica si el elemento se encuentra o no en la secuencia

- `secuencia[i]`

  Corresponde al valor de la secuencia en la posición `i`, comenzando desde 0. Si se utilizan números negativos, se puede acceder a los elementos desde el último `(-1)` hasta el primero `(-len(secuencia))`. En el caso de las tuplas o cadenas (inmutables) sólo puede usarse para obtener el valor, mientra que en las listas (mutables) puede usarse también para modificar su valor.

- `secuencia[i:j:k]`

  Permite obtener un segmento de la secuencia, desde la posición `i` inclusive, hasta la posición `j` exclusive, con paso `k`.

  En el caso de que se omita `i`, se asume 0. En el caso de que se omita `j`, se asume `len(secuencia)`. En el caso de que se omita `k`, se asume 1. Si se omiten todos, se obtiene una copia completa de la secuencia.

- `lista.append(valor)`

  Agrega un elemento al final de la lista.

- `lista.insert(posicion, valor)`

  Agrega un elemento a la lista, en la posición `posicion`.

- `lista.remove(valor)`

  Quita de la lista la primera aparición de elemento, si se encuentra. De no encontrarse en la lista, se produce un error.

- `lista.pop()`

  Quita el elemento del final de la lista, y lo devuelve. Si la lista está vacía, se produce un error.

- `lista.pop(posicion)`

  Quita el elemento que está en la posición indicada, y lo devuelve. Si la lista tiene menos de `posicion + 1` elementos, se produce un error.

- `lista.index(valor)`

  Devuelve la posición de la primera aparición de valor. Si no se encuentra en la lista, se produce un error.

- `sorted(secuencia)`

  Devuelve una lista nueva, con los elementos de la secuencia ordenados.

- `lista.sort()`

  Ordena la misma lista.

- `cadena.split(separador)`

  Devuelve una lista con los elementos de cadena, utilizando `separador` como separador de elementos. Si se omite el separador, toma todos los espacios en blanco como separadores.

- `separador.join(lista)`

  Genera una cadena a partir de los elementos de `lista`, utilizando `separador` como unión entre cada elemento y el siguiente.