

Universidad Nacional de la Patagonia San Juan Bosco

Laboratorio de Investigación en Informática



Conceptos Avanzados de Programación Python

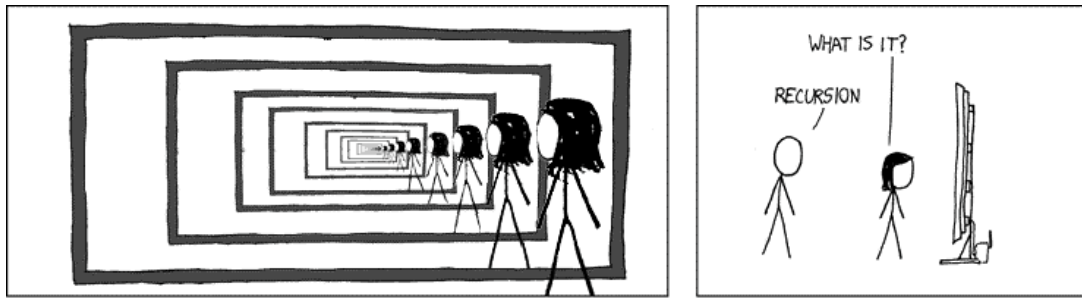
Noviembre 2023

Parte 3

Índice

1. Recursión	2
2. ¿Dónde (y cómo) se mantienen las computaciones intermedias?	5
3. Programación orientada por objetos	7
4. Definiendo nuevas clases	8
5. Atributos de la clase, atributos “por defecto”	10
6. Métodos “especiales”	11
7. Métodos para comparar objetos	13
8. Resumen de OOP	14
9. Alteración del flujo natural del control	15
10. Manejo de excepciones	17
11. Algunas de las excepciones más comunes	19
12. Google Colab	21

1. Recursión



Una función se denomina recursiva cuando puede invocarse a sí misma durante su ejecución. En este caso se denomina recursiva directa (existe la recursión indirecta cuando una función invoca a otra y viceversa). La definición de funciones recursivas tiene su sustento matemático en el principio de inducción (ver https://es.wikipedia.org/wiki/Induccion_matematica):

Dado un número entero e , que tiene la propiedad P (caso base), y pudiendo demostrar que para cualquier número entero n , si n tiene la propiedad P entonces $n + 1$ también la tiene (caso inductivo), entonces todos los números enteros a partir de e , tienen la propiedad P .

Un ejemplo típico de este principio es la *definición inductiva* de funciones, por ejemplo la función factorial:

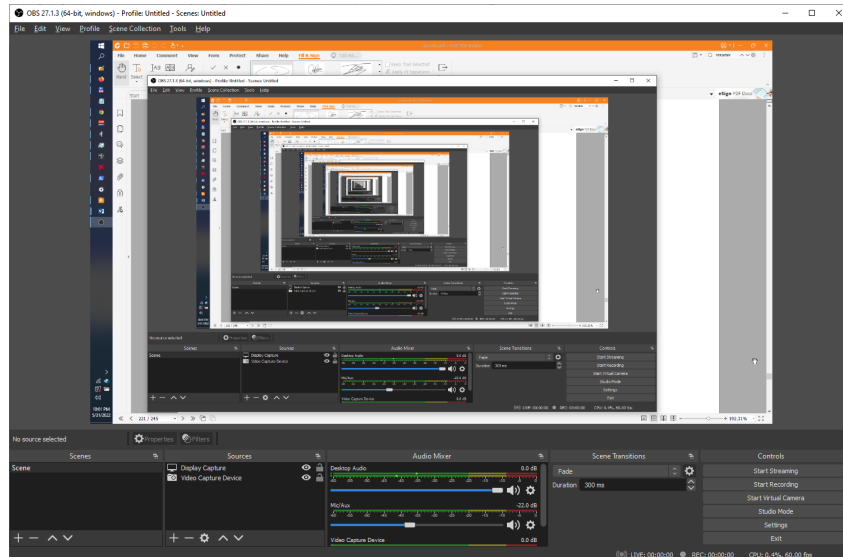
$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n \cdot (n - 1)! & \text{caso contrario} \end{cases}$$

Esta definición lleva a un “correlato” algorítmico inmediato:

```
def factorial(n) :  
    if n == 0 : return 1  
    else : return n * factorial(n - 1)
```

Evidentemente si la invocación recursiva fuese incondicional, entonces entraríamos en un lazo infinito, por lo que tenemos que conocer y tener en cuenta los *principios de diseño* adecuados para que una función recursiva no caiga en este riesgo:

1. La función debe tener por lo menos un caso base (no recursivo) y nunca debe ejecutarse una llamada recursiva si corresponde ejecutarse un caso base.
2. La llamada recursiva debe hacerse sobre argumentos que acerquen y garanticen que se llegue al caso base (i.e., los argumentos están “más cerca” del caso base que los parámetros). Puede haber más de una posible llamada recursiva.
3. La complejidad con la cual los parámetros recibidos se transforman en los argumentos en la llamada recursiva debe ser menor que el cálculo recursivo en sí mismo.



En el ejemplo del factorial, la “utilidad” de utilizar recursión es dudosa, la función iterativa es también sencilla y posiblemente más eficiente (aunque tiene algunos defectos):

```
def factorial(n) :
    fact = 1
    for num in range(1, n+1) : fact *= num
    return fact
```

En algunos casos hay versiones recursivas e iterativas con paridad de virtudes y defectos, en otros hay alguna versión que es claramente mejor que la otra, etc.

```
def PotEntera(b,n) :
    if b == 0 : return 0
    else :
        if n == 0 : return 1
        elif n < 0 : return 1 / PotEntera(b,-n)
        elif n % 2 == 0 :
            p = PotEntera(b, n // 2)
            return p * p
        else :
            p = PotEntera(b, (n - 1) // 2)
            return p * p * b
```

Comparar (y razonar acerca de) la eficiencia de la función anterior respecto de la siguiente:

```
def fib(n):
    if n == 0 or n == 1 : return n
    else : return fib(n - 1) + fib(n - 2)
```

Un ejemplo revisitado

Supongamos que queremos retomar el ejemplo de la sumatoria de la serie visto en clase, pero queremos simplificar las fracciones.

```
def suma_frac(frac1,frac2):
    dic_suma_frac = {}
    dic_suma_frac["den"] = frac1["den"] * frac2["den"]
    dic_suma_frac["num"]=frac2["den"]*frac1["num"]+frac1["den"]*frac2["num"]
    simplificar(dic_suma_frac)
    return(dic_suma_frac)
```

Para eso utilizamos la definición recursiva de máximo común divisor entre dos enteros:

```
mcd(a, b) = b si a es divisible por b
mcd(a, b) = mcd(b, a mod b)
```

```
def simplificar(frac) :
    n = mcd(frac["num"], frac["den"])
    frac["num"], frac["den"] = frac["num"]//n, frac["den"]//n

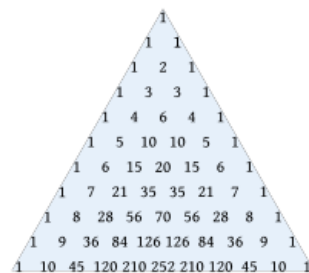
def mcd(a,b) :
    if a%b == 0 : return b
    else : return(mcd(b, a%b))
```

Observar que `simplificar` simplifica la fracción como efecto colateral (lo cual puede ser bueno o no, dependiendo del uso esperado).

Otro ejemplo revisitado

Recordamos cuando construimos el triángulo de Pascal (los números combinatorios). Llamamos `comb(i,j)` al número de combinaciones de i elementos en j lugares (i es “las columnas” y j “las filas” en caso de escribir estos números en forma encolumnada).

Triángulo de Pascal



Una propiedad que observamos que en caso general se cumple es la siguiente:

$$\text{comb}(i,j)=\text{comb}(i-1,j-1)+ \text{comb}(i,j-1).$$

Esta propiedad (recursiva) deja de cumplirse “fuera” del triángulo (si la columna es cero $i==0$, o si estamos “en la diagonal” $i==j$), en ambos casos el valor es uno (casos base). Observar que esta definición, al igual que la de Fibonacci que vimos más arriba tienen *complejidad exponencial*: la cantidad de llamadas recursivas crece exponencialmente con

```
def comb(i,j) :
    if i==j or i==1 : return 1
    else : return(comb(i-1, j-1) + comb(i, j-1))
```

los parámetros de llamada. Vemos entonces una situación bastante “dramática” en la recursión: si las llamadas recursivas particionan el problema a una fracción (la mitad, por ejemplo, como en el caso de la potencia), entonces la complejidad del cálculo crece con el logaritmo (base dos) de los parámetros (calcular la potencia 100 requiere aproximadamente 8 llamadas recursivas), mientras que si la partición es de a un elemento, la complejidad crece con la potencia de dos.

2. ¿Dónde (y cómo) se mantienen las computaciones intermedias?

Si miramos el siguiente segmento de código y su ejecución podemos comprobar que, pese a tener el mismo nombre, la variable `x` de la función `f` y la variable `x` de la función `g` no tienen nada que ver una con otra. Se refieren a objetos distintos, y modificar una no modifica a la otra.

```
def f(x):
    b = 20
    print("En f, x vale", x)

def g(x):
    b = 45
    print("En g, antes de llamar a f, x vale", x)
    f(b)
    x = 10
    print("En g, después de llamar a f, x vale", x)
```

Este sería un ejemplo de salida en la ejecución de `g()`:

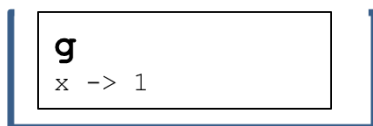
```
>>> g(1)
En g, antes de llamar a f, x vale 1
En f, x vale 45
En g, después de llamar a f, x vale 10
```

Este comportamiento lo aceptamos desde un comienzo, pero nunca nos detuvimos a pensar por qué sucede, y ahora con las funciones recursivas tenemos que comprender mejor cómo se ejecutan las llamadas a funciones para comprender mejor cuál es la razón de este comportamiento. En Python, cada contexto de referenciamiento está representado por un **frame** (marco), que contiene toda la información necesaria para desambiguar los valores u objetos a los que están referenciando cada uno de los identificadores de dicho contexto. Al comenzar la ejecución del intérprete, el mismo tiene un frame que es el

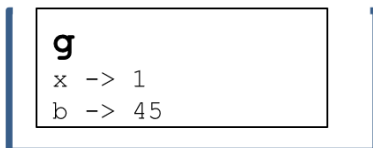
“fondo” o punto de apoyo sobre el cual se van colocando los frames de las diferentes funciones a medida que éstas se van llamando. El marco de cada función contiene los nombres de cada una de las variables y parámetros, y el “lugar” para asociar a estos los valores u objetos correspondientes. Veamos en nuestro ejemplo:



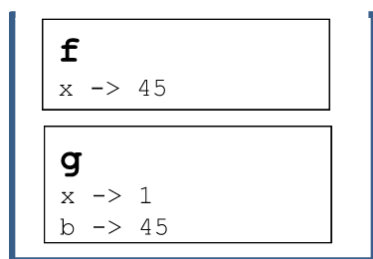
Estado inicial antes de llamar a la función **g**: no hay frames ni variables declaradas



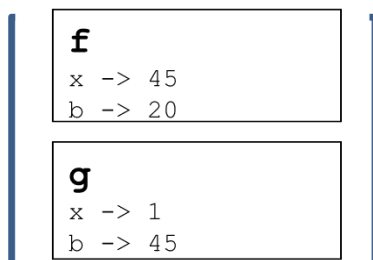
Se produce el llamado a **g**: se asigna el valor 1 al parámetro **x** (el valor del argumento en la llamada).



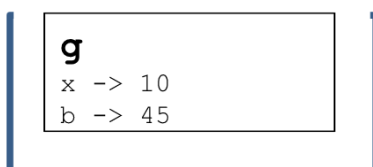
Se crea una nueva variable (local a **g**) y se le asigna su valor.
Se ejecuta el print.



Se efectúa el llamado a la función **f(b)** desde **g**. **f** recibe a este argumento como parámetro **x** (local a **f**).



Se crea una nueva variable (local a **f**). Esta variable tiene un nombre igual a otra (y por lo tanto la “pisa”).
Se ejecuta el print.



f termina (y su frame desaparece!). Se asigna un nuevo valor a **x** y se ejecuta el print.

Si a su vez la función **f** llamase a otra función **h** durante su ejecución, el frame de **h** se depositaría por encima de los frames existentes. Es fácil ver que los frames se gestionan como una pila: el último en ser creado se deposita encima de los demás, y cuando su función asociada termina, es retirado. Por ello a esta estructura se la denomina “*pila de activaciones*”.

También es importante ver que el nombre de la función es irrelevante en cada frame, solo importa el orden en el cual los frames se crean y eliminan. Si una función es recursiva, y durante su ejecución se invoca a sí misma, entonces habrán dos (o más) frames de dicha función, que representan dos *instancias distintas* de su activación.

Cuidado especial hay que tener cuando se llama a una función con un argumento mutable (ver el ejemplo de la función `simplificar` más arriba). En este caso, en la función invocada no se crea un objeto nuevo asociado al valor recibido por parámetro, sino que el parámetro pasa a ser una **referencia** al mismo objeto que el argumento recibido. Por lo tanto cualquier cambio a dicho parámetro dentro de la función, modifica el valor del argumento recibido fuera de ella (por efecto **colateral**).

3. Programación orientada por objetos

Los objetos son estructuras que permiten diseñar y desarrollar algoritmos y programas, con ventajas que facilitan aún más lo hecho con las estructuras de datos y las funciones que hemos visto hasta ahora. Como tal, la Programación Orientada por Objetos (OOP) es un **“paradigma”** que involucra algunos cambios importantes respecto de nuestra forma de programar.

En realidad ya veníamos usando objetos en Python (sin mencionarlo explícitamente). Sin embargo, la manera de hacerlo no estaba establecida formalmente. Por ejemplo, cuando utilizamos `lista.pop()`, estamos llamando al **método** `pop` sobre un objeto de la clase `list`. Los valores asociados a ese objeto (su longitud, por ejemplo) se denominan atributos.

La OOP introduce o modifica la terminología establecida, dado que al realizarse un cambio de paradigma es necesario cambiar la forma de nombrar a los elementos que se aprendieron en el paradigma anterior.

En Python los **tipos** son **clases** predefinidas (aunque no en todos los lenguajes de programación es así), y un **objeto** es una instancia de una **clase**. En general, podemos decir que una clase es una forma específica de organizar los **atributos** y los **métodos** a utilizar sobre dichos atributos u otros datos. Los objetos tienen **estado** y **comportamiento**, ya que los valores que tengan sus atributos determinan la situación actual de esa instancia, y los métodos definidos en una clase determinan cómo se va a **comportar** un objeto de una clase dada.

En el caso de los tipos básicos (números, por ejemplo) esta superestructura parece excesiva y por eso nunca la mencionamos hasta ahora. Por otro lado, tampoco es una buena idea que un lenguaje tenga dos grupos distintos de elementos (tipos “simples” y clases).

Nuestro objetivo ahora es poder definir nuestras propias clases, en cuya especificación se indican cuáles son los atributos y métodos que van a tener los objetos que sean de esa

clase. A partir de una clase es posible crear distintos objetos que pertenecen a esa clase, pero también (ya veremos cómo) crear clases más complejas a partir de incorporar la definición de otras clases más simples.

4. Definiendo nuevas clases

Queremos definir nuestra clase que represente un punto en el plano. Lo primero que debemos notar es que existen varias formas de representar un punto en el plano, por ejemplo, coordenadas polares o coordenadas Cartesianas. En esta primera implementación, optaremos por utilizar la representación de coordenadas Cartesianas, e iremos implementando las operaciones a medida que las vayamos necesitando. En primer lugar, creamos una **clase** `Punto2D` que simplemente almacena las coordenadas.

```
class Punto2D :
    """Representación de un punto en el plano en coordenadas Cartesianas"""

    def __init__(self, x, y) :
        """Constructor del Punto2D. x e y deben ser numéricos"""

        print('Creando un objeto Punto2D con x={} y={}'.format(x,y))
        self.x = x
        self.y = y
```

(El print es totalmente supérfluo, lo agregamos solamente para poder ver cuándo se ejecuta el constructor).

En la primera línea de código indicamos que vamos a crear una nueva clase, llamada `Punto2D`. Por convención, en los nombres de las clases definidas por el programador se escribe cada palabra del nombre con la primera letra en mayúsculas.

Además, definimos uno de los métodos especiales, `__init__` el **constructor** de la clase. Este método se llama cada vez que se crea una nueva instancia de la clase, recibiendo como primer parámetro a la instancia misma sobre la que está trabajando (sería una “auto-referencia”). Por convención a ese primer parámetro se lo llama `self` (“a sí mismo”) para que se entienda que la definición actúa sobre esa instancia particular.

```
>>> p = Punto2D(4,5)
Creando un objeto Punto2D con x=4 y=5
>>> p
<__main__.Punto2D object at 0x000001EF57B69D20>
>>> p.x
4
>>> type(p)
<class '__main__.Punto2D'>
```

Hemos creado una clase Punto2D que permite guardar valores x e y. Sin embargo, por más que en “la documentación” se indique que los valores deben ser numéricos, el código no impide que a x o y se les asigne un valor cualquiera, no numérico.

```
>>> q=Punto2D('pepe', (1,2,3))
Creando un objeto Punto2D con x=pepe y=(1, 2, 3)
>>> q.y
(1, 2, 3)
```

Si queremos impedir que esto suceda, debemos agregar validaciones al constructor, Verificaremos que los valores pasados para x e y sean numéricos, utilizando una función `validar_numero`.

```
class Punto2D :
    def __init__(self, x, y) :
        def validar_numero(x) : return isinstance(x, (int, float))

        if validar_numero(x) and validar_numero(y) :
            print('Creando un objeto Punto2D con x={} y={}'.format(x,y))
            self.x = x
            self.y = y
        else : print('valor(es) no válido(s)')
```

Observar que el objeto es creado aunque no se le asignen valores a sus atributos:

```
>>> p = Punto2D(4,5)
Creando un objeto Punto2D con x=4 y=5
>>> q = Punto2D('x',4)
valor(es) no válido(s)
>>> q
<__main__.Punto2D object at 0x00000284E3C0B9D0>
>>> q.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Punto2D' object has no attribute 'x'
```

Hasta ahora hemos creado una clase Punto2D que permite construir objetos con un par de atributos, que deben ser numéricos, pero no podemos operar con sus valores (salvo que vayamos atributo por atributo). Queremos definir métodos adicionales, por ejemplo, poder calcular la distancia entre dos puntos. Para ello definimos un nuevo método `distancia` que se aplica sobre un objeto Punto2D y recibe por parámetro otro punto entre el cual se quiere calcular la distancia.

```
def distancia(self, otro) :
    """Devuelve la distancia entre ambos puntos."""
    dx = self.x - otro.x
    dy = self.y - otro.y
    return (dx * dx + dy * dy) ** 0.5
```

```
>>> p=Punto2D(3,4)
>>> q=Punto2D(1,2)
>>> p.distancia(q)
2.8284271247461903
>>> q.distancia(p)
2.8284271247461903
>>> q.distancia(q)
0.0
```

Podemos ver, sin embargo, que la operación para calcular la distancia incluye la operación de restar dos puntos y la de obtener la norma de un vector. Sería deseable incluir también estas dos operaciones dentro de la clase. Agregaremos, entonces, el método para restar dos puntos:

```
def restar(self, otro) :
    """Devuelve la resta entre dos puntos."""
    return Punto2D(self.x - otro.x, self.y - otro.y)
```

Si bien la resta entre dos puntos no es un nuevo punto (por qué?) por el momento este método devuelve una nueva instancia de Punto2D, en lugar de modificar las instancias self u otro. A continuación definimos el método para calcular la norma del vector que se forma uniendo un punto con el origen.

```
def norma(self) :
    """Norma del vector que va desde el origen hasta el punto. """
    return (self.x * self.x + self.y * self.y) ** 0.5
```

En base a estos dos métodos podemos ahora volver a escribir el método distancia para que aproveche el código ambos

```
def distancia(self, otro) :
    """Devuelve la distancia entre ambos puntos."""
    return self.restar(otro).norma()
```

En definitiva, hemos definido tres métodos en la clase Punto2D, que nos sirven para calcular restas, normas de vectores al origen, y distancias entre puntos.

5. Atributos de la clase, atributos “por defecto”

En muchos contextos, **todas** las instancias de una clase (los objetos creados que pertenecen a la misma) poseen atributos genéricos de la clase.

```
>>> p = Punto2D(5, 7)
>>> q = Punto2D(2, 3)
>>> r = p.restar(q)
>>> (r.x, r.y)
(3, 4)
>>> r.norma()
5.0
>>> q.distancia(r)
1.41421356237
```

```
class Perro:

    especie = "cánido"

    def __init__(self, nom, ed, rz):
        self.nombre = nom
        self.edad = ed
        self.raza = rz

>>> Mascota_de_Marina = Perro('Nina',8,'Toy')
>>> Mascota_de_Marina.edad
8
>>> Mascota_de_Marina.__class__.especie
'cánido'
```

Igualmente a lo que ocurre con los parámetros de las funciones, los métodos pueden tener valores por defecto

```
def __init__(self, nom, ed, rz='mestizo'):
    self.nombre = nom
    self.edad = ed
    self.raza = rz

>>> Mascota_de_Klaus = Perro('Bobby',0.5)
>>> Mascota_de_Klaus.raza
'mestizo'
```

6. Métodos “especiales”

El método `__init__` debe estar en toda clase, y se invoca cuando se intenta crear un objeto de esa clase. En ese sentido es “especial” dado que posee mecanismos (y un nombre) que no son exactamente los mismos que para otros métodos. Del mismo modo, hay otros métodos que son muy prácticos y necesarios en varios contextos. Uno de ellos es indispensable cuando queremos “imprimir” el estado de un objeto:

```
>>> p
<__main__.Perro object at 0x00000279D11BAFB0>
>>> str(p)
'<__main__.Perro object at 0x00000279D11BAFB0>'
```

Si queremos que nuestra clase incluya la posibilidad de imprimir el estado de sus objetos, entonces **sobrecargamos** la función `str()` del siguiente modo:

```
def __str__(self) :
    return('Perro({},{},{})'.format(self.nombre,self.edad,self.raza))
```

Muchas de las funciones provistas por Python, que ya hemos utilizado, como `str`, `len` o `help`, invocan internamente a los métodos de los objetos correspondientes. Es decir que la función `str` internamente invoca al método `__str__` del objeto que recibe como parámetro.

Esto muestra que el método `__init__` es el que se ejecuta en la creación de un objeto cuando se lo asignamos a una variable, y que estos “métodos especiales” están definidos por **sobrecarga** para las clases predefinidas. Claramente hay otros métodos sobrecargados, por ejemplo el `+` asociado a números, strings y colecciones. El método `__repr__` nos permite acceder a la “representación interna” de un objeto.

Volviendo a nuestros Punto2D, quisiéramos sobrecargar la suma y la resta:

```
>>> p1=Punto2D(1,2)
>>> p2=Punto2D(4,3)
>>> p1-p2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -:
'Punto2D' and 'Punto2D'
```

```
def __add__(self, otro) :
    """Devuelve la suma de ambos puntos."""
    return Punto2D(self.x + otro.x, self.y + otro.y)

def __sub__(self, otro):
    """Devuelve la resta de ambos puntos."""
    return Punto2D(self.x - otro.x, self.y - otro.y)

>>> p=Punto2D(4,3)
>>> q=Punto2D(2,5)
```

```

>>> p-q
Punto2D(2,-2)
>>> p+p
Punto2D(8,6)
>>> r=p+p-q
>>> r
Punto2D(6,1)

```

7. Métodos para comparar objetos

Para resolver comparaciones entre puntos (por ejemplo por su norma), será necesario definir algunos métodos especiales que permiten comparar objetos. En particular, cuando se quiere que los objetos puedan ser ordenados, es suficiente con definir el método `__lt__` (less than), que corresponde al operador matemático de comparación $<$. El método `__lt__` recibe dos parámetros, `self` y `otro` y debe devolver `True` si `self` es comparativamente “menor” a otro de acuerdo a nuestra definición.

```

def __lt__(self, otro):
    """Compara dos Puntos2D según sus normas."""
    return self.norma() < otro.norma()

```

Una vez que tenemos definido el `__lt__`, esto nos permite ordenar listas de objetos:

```

>>> p=Punto2D(1,2)
>>> q=Punto2D(3,2)
>>> r=p-q
>>> s=p+p-q
>>> w=Punto2D(8,-8)
>>> lista=[p,q,r,s,r,q,p,w]
>>> lista
[Punto2D(1,2), Punto2D(3,2), Punto2D(-2,0), Punto2D(-1,2), Punto2D(-2,0),
Punto2D(3,2), Punto2D(1,2), Punto2D(8,-8)]
>>> lista.sort()
>>> lista
[Punto2D(-2,0), Punto2D(-2,0), Punto2D(1,2), Punto2D(-1,2), Punto2D(1,2),
Punto2D(3,2), Punto2D(3,2), Punto2D(8,-8)]

```

También es posible utilizar el flag `key` en el método `sort`. Si queremos ordenar por el valor de `y`, por ejemplo, utilizamos una **función lambda**.

```
>>> lista.sort(key=lambda p:p.y)
>>> lista
[Punto2D(8,-8), Punto2D(-2,0), Punto2D(-2,0), Punto2D(1,2), Punto2D(-1,2),
Punto2D(1,2), Punto2D(3,2), Punto2D(3,2)]
```

Las funciones lambda son “funciones anónimas” (hace falta una función, pero no hace falta definirla, darle un nombre, etc.). Tienen una sintaxis y comportamiento similar a una función “normal” pero no requieren toda la burocracia de la definición usual.

También se puede sobrecargar el operador de igualdad `==` utilizando `__eq__`.

8. Resumen de OOP

```
class NombreClase:
```

Indica que se comienza a definir una clase con el nombre indicado.

```
def __init__(self, ...):
```

Define el *método constructor* de la clase. En general, dentro del constructor se establecen los valores iniciales (defaults) de todos los atributos.

```
variable = NombreClase(...)
```

Crea un nuevo objeto como instancia de la clase. Los parámetros que se ingresen serán pasados a la constructora, luego del parámetro especial `self`.

```
variable.atributo
```

Permite obtener o modificar el valor de un atributo de la instancia.

```
def metodo(self, ...)
```

El primer parámetro de cada método de una clase es una referencia a la instancia sobre la que va a operar el método. Se lo llama por convención `self`, pero puede tener cualquier nombre.

```
variable.metodo(...)
```

Invoca al `metodo` de la clase de la cual `variable` es un objeto. El primer parámetro que se le pasa a `metodo` (usualmente `self`) será instanciado a `variable`.

```
def __str__(self):
```

Método especial que debe devolver una cadena de caracteres, con la representación “informal” de la instancia. Se invoca al hacer `str(variable)` o `print(variable)`.

```
def __repr__(self):
```

Método especial que debe devolver una cadena de caracteres, con la representación “formal” de la instancia. Se invoca al hacer `repr(variable)`.

```
def __add__(self, otro):, def __sub__(self, otro):
```

Métodos especiales para sobrecargar los operadores `+` y `-` respectivamente. Reciben las dos instancias sobre las que se debe operar, debe devolver una nueva instancia con el resultado.

```
def __lt__(self, otro):
```

Método especial para permitir la comparación de objetos mediante sobrecarga de los operadores `<` y `>`. Recibe las dos instancias a comparar. Debe devolver `True` si `self` es comparativamente “menor” a otro de acuerdo a cómo se defina en la clase.

```
def __le__(self, otro):
```

Método especial para permitir la comparación de objetos mediante la sobrecarga de los operadores `<=` y `>=`. Devuelve `True` si `self` es comparativamente “menor o igual” a otro.

```
def __eq__(self, otro):
```

Método especial para permitir la comparación de objetos mediante sobrecarga de los operadores `==` y `!=`. Devuelve `True` si `self` y otro son comparativamente “iguales”.

9. Alteración del flujo natural del control

Si bien la gran mayoría de los problemas de programación pueden resolverse con las estructuras de control que ya hemos visto, en algunos casos es imposible o demasiado forzado expresar el comportamiento esperado de un programa solo con ellas. Por dicho motivo se introducen algunos mecanismos adicionales. `*Break`

En varios ejemplos vimos que es necesario “repetir” alguna sentencia para expresar una estructura de control dada. Por ejemplo un programa que pide entrada a la usuaria y luego ejecuta una tarea y regresa, o bien se termina la ejecución:

Parece más sensato preguntar al comienzo del `while`, y si la respuesta es salir, entonces abandonar el `while`, pero no contamos con un mecanismo para hacerlo.


```

Flag = input(" bla bla ")
while Flag :
    if Flag == "N" : hacer algo
    elif Flag == "L" : hacer otra cosa
    ...                etc.
    Flag = input(" bla bla de nuevo ")

```

```

while Pos < len(lista_enteros) and not Lo_Encontre :
    if lista_enteros[Pos] == Buscado :
        Lo_Encontre = True
    else:
        Pos = Pos + 1

```

Otro ejemplo similar ocurrió en la búsqueda secuencial, donde vamos recorriendo una lista de a un elemento hasta encontrar el buscado o terminar.

Aquí también parece sensato no estar preguntando en cada paso si “lo encontré” sino más bien abandonar el while cuando ello ocurra. En estos y otros casos nos sirve el mecanismo **break** como parte de un ciclo, que como su nombre indica “rompe” la ejecución del ciclo y lleva el control a la sentencia inmediata posterior. Nuestros ejemplos “mejorados” con el uso del **break** quedarían así:

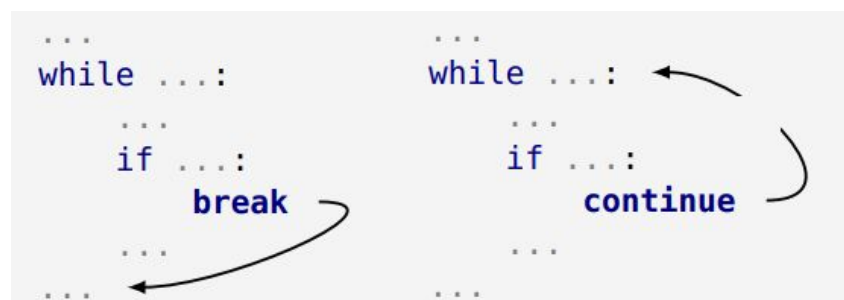
```

while True :
    Flag = input(" bla bla única vez ")
    if Flag == "N" : hacer algo
    elif Flag == "L" : hacer otra cosa
    elif Flag == "Salir" : break

while Pos < len(lista_enteros) :
    if lista_enteros[Pos] == Buscado : break
    else: Pos = Pos + 1

```

En breve: cuando durante la ejecución del cuerpo de un ciclo **for** o **while** se llega a una instrucción **break**, la ejecución abandona el ciclo inmediatamente. Similarmente existe la sentencia **continue**, en cuyo caso la ejecución abandona el resto del ciclo que se está ejecutando y se pasa inmediatamente al ciclo siguiente.



10. Manejo de excepciones

La otra situación en la cual es deseable interrumpir el flujo natural del control es cuando ocurre algo excepcional (archivo no encontrado, división por cero, dato mal conformado, etc.) en cuyo caso lo que normalmente ocurre es que la excepción nos interrumpe el programa o nos genera un mensaje de error indeseado. Además no siempre podemos predecir en qué contexto ocurrirá la excepción. Podemos tener una función F que potencialmente genera una división por cero, la cual puede ser a su vez llamada desde otras funciones. Si la función A, llama a la función B y ésta a F y en F ocurre la excepción, si F no gestiona la excepción entonces se interrumpe y el control retorna a B. Si B no maneja la excepción, se interrumpe y retorna a A, etc.

Para evitar esta cadena de malas posibilidades, tenemos la sentencia `try`, que permite englobar la o las sentencias que pudiesen generar una excepción, la cual en caso de ocurrir es gestionada por otro grupo de sentencias englobadas dentro de la sentencia `except`.

```
def reciproco(n):
    try: return 1/n
    except: return 999999999999999999

while True:
    n = input('Ingresar dato ')
    if n=='': break
    print('El recíproco de {} es {}'.format(int(n), reciproco(int(n))))
```

```
Ingresar dato 1
El recíproco de 1 es 1.0
Ingresar dato 2
El recíproco de 2 es 0.5
Ingresar dato 0
El recíproco de 0 es 999999999999999999
Ingresar dato
```

Dado que dentro de un mismo bloque `try` pueden producirse excepciones de distinto tipo, es posible definir un bloque `except` para cada caso. Esto se hace especificando el nombre de la excepción que se pretende capturar. Si bien luego de un bloque `try` puede haber varios bloques `except`, se ejecutará a lo sumo uno de ellos.

```
try:
    # aquí está el código que puede generar excepciones
except IOError:
    # excepción IOError es de entrada/salida
except ZeroDivisionError:
    # excepción ZeroDivisionError ya se sabe qué es
except:
    # en caso que se haya producido una excepción diferente
```

En caso de necesitarlo, podemos atrapar el tipo de excepción generada (por ejemplo, para devolverlo a una función llamadora y que ésta determine cómo manejar el error).

```
def reciproco2(n):
    try:
        return 1/n
    except Exception as ex:
        print('Ocurrió una excepción de tipo {}'.format(type(ex).__name__))

>>> reciproco2(1)
1.0
>>> reciproco2(0)
Ocurrió una excepción de tipo ZeroDivisionError
>>> reciproco2('a')
Ocurrió una excepción de tipo TypeError
```

En otros contextos la situación es “semánticamente” inadecuada, entonces nosotros deseamos específicamente generar una excepción (por ejemplo para que en otra parte del código esa excepción esté “atrapada” por un `try-except`).

```
class Triangulo:
    def __init__(self, a, b, c):
        lados = [a, b, c]
        lados.sort()
        if lados[0]+lados[1] < lados[2]:
            raise ValueError('Triángulo mal conformado!')
        self.a = a
        self.b = b
        self.c = c
```

Con esta definición de la clase, el método constructor no creará un triángulo si se ejecuta el `raise` (se abandona la función sin terminar de ejecutarla). De esa manera podemos decidir realizar o no la construcción del triángulo desde el mismo método constructor y sin tener que validar los parámetros antes:

```
L1 = int(input('Ingresar lado uno del triángulo: '))
L2 = int(input('Ingresar lado dos del triángulo: '))
L3 = int(input('Ingresar lado tres del triángulo: '))
try:
    T = Triangulo(L1,L2,L3)
except Exception as ex:
    print('Ocurrió una excepción de tipo {}'.format(type(ex).__name__))
```

11. Algunas de las excepciones más comunes

```
exception ArithmeticError
```

La clase base para las excepciones predefinidas que se generan para varios errores aritméticos: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

```
exception BufferError
```

Se genera cuando en un buffer no se puede realizar una operación dada.

```
exception AttributeError
```

Se genera cuando se produce un error en una referencia de atributo o la asignación falla. (Cuando un objeto no admite referencias de atributos o asignaciones de atributos en absoluto, se genera `TypeError`)

```
exception EOFError
```

Se genera cuando la función `input()` alcanza una condición de fin de archivo (EOF) sin leer ningún dato (en cambio `io.IOBase.read()` y `io.IOBase.readline()` retornan una cadena vacía cuando llegan a EOF).

```
exception FloatingPointError
```

Excepción desactualizada, para errores numéricos, pero no se usa en las nuevas versiones.

```
exception ImportError
```

Se genera cuando la instrucción `import` tiene problemas al intentar cargar un módulo. También se produce cuando la `from list` en `from ... import` tiene un nombre que no se puede encontrar.

```
exception IndexError
```

Se genera cuando un subíndice de secuencia está fuera del rango. (Los índices de la rebanada son truncados silenciosamente para caer en el intervalo permitido; si un índice no es un entero, se genera `TypeError`.)

```
exception KeyError
```

Se genera cuando no se encuentra una clave de asignación (diccionario) en el conjunto de claves existentes (mapa).

```
exception KeyboardInterrupt
```

Se genera cuando el usuario pulsa la tecla de interrupción (normalmente **Control-C** o **Delete**). Durante la ejecución, se realiza una comprobación de interrupciones con regularidad.

```
exception MemoryError
```

Se genera cuando una operación se queda sin memoria pero la situación aún puede ser recuperada (eliminando algunos objetos). El valor asociado es una cadena que indica que tipo de operación (interna) se quedó sin memoria.

```
exception NameError
```

Se genera cuando no se encuentra un nombre local o global. Esto se aplica sólo a nombres no calificados. El valor asociado es un mensaje de error que incluye el nombre que no se pudo encontrar. El atributo `name` se puede establecer utilizando un argumento de solo palabra clave para el constructor. Cuando se establece, representa el nombre de la variable a la que se intentó acceder.

```
exception OSError([arg], errno, strerror[, filename[, winerror[, filename2]]])
```

Esta excepción se produce cuando una función del sistema retorna un error relacionado con el sistema, que incluye fallas de E/S como `file not found` o `disk full` (no para tipos de argumentos ilegales u otros errores incidentales).

```
exception OverflowError
```

Se genera cuando el resultado de una operación aritmética es demasiado grande para ser representado.

```
exception RecursionError
```

Esta excepción se eleva cuando el intérprete detecta que se excede la profundidad máxima de recursión.

```
exception RuntimeError
```

Se genera cuando se detecta un error que no corresponde a ninguna de las otras categorías. El valor asociado es una cadena que indica exactamente qué salió mal.

```
exception SyntaxError(message, details)
```

Se genera cuando el analizador encuentra un error de sintaxis. El atributo `str()` de la instancia de excepción retorna solo el mensaje de error.

```
exception IndentationError
```

Clase base para errores de sintaxis relacionados con sangría incorrecta.

```
exception SystemError
```

Se genera cuando el intérprete encuentra un error interno, pero la situación no parece tan grave como para abandonar la ejecución. El valor asociado es una cadena que indica qué salió mal (a bajo nivel).

```
exception TypeError
```

Se genera cuando una operación o función se aplica a un objeto de tipo inapropiado. El valor asociado es una cadena que proporciona detalles sobre la falta de coincidencia de tipos.

```
exception ValueError
```

Se genera cuando una operación o función recibe un argumento que tiene el tipo correcto pero un valor inapropiado.

```
exception ZeroDivisionError
```

Se genera cuando el segundo argumento de una operación de división o módulo es cero. El valor asociado es una cadena que indica el tipo de operandos y la operación.

12. Google Colab

Se basa en el uso de Jupyter Notebooks (cuadernos), que es un entorno de programación inicialmente pensado para IDEs como Conda.

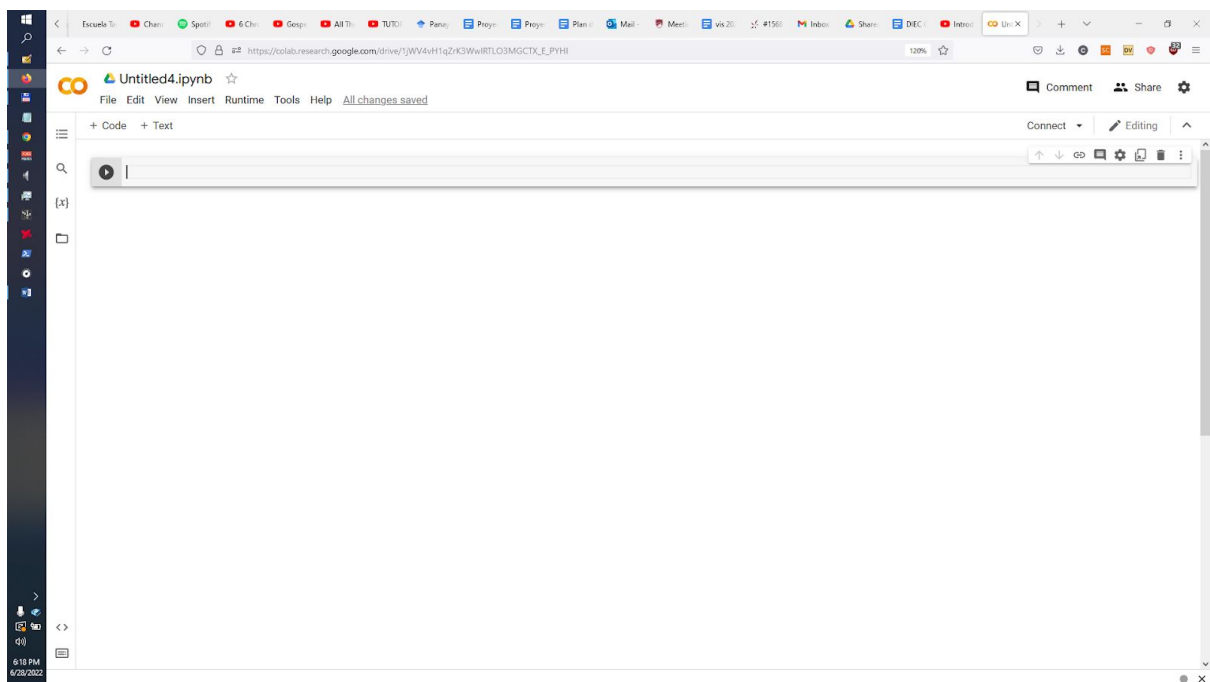
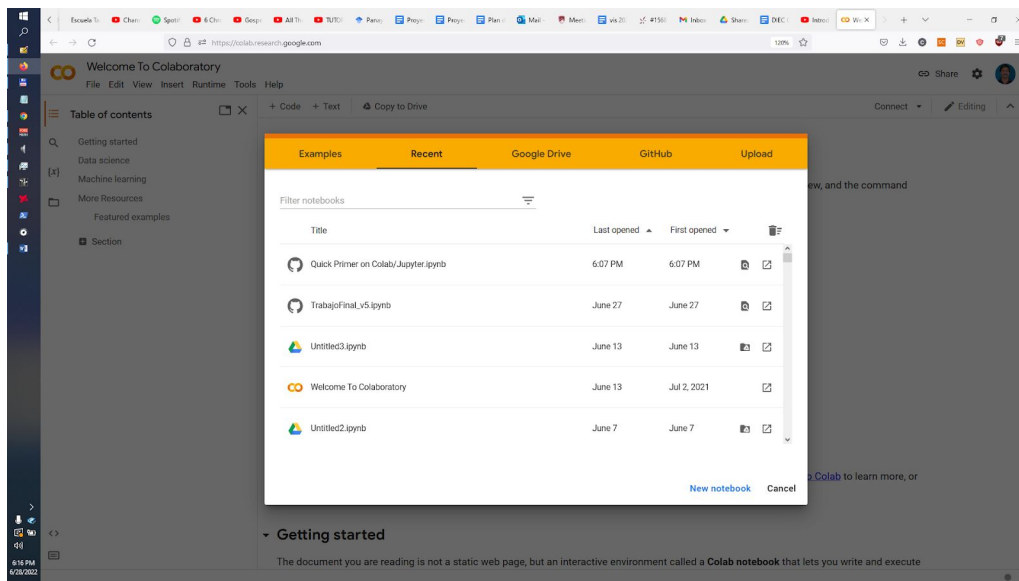
Las Notebooks (NB) permiten escribir texto formateado (en lenguaje Markdown) junto con celdas de código, que puede ser en Julia, Python o R (JU-PYT-eR).

Google tomó esta iniciativa y formato y la adaptó para poder correr en sus servidores en la nube. Simplemente ingresando a <http://colab.research.google.com/> (teniendo una cuenta de Gmail) es posible comenzar a utilizar el entorno.

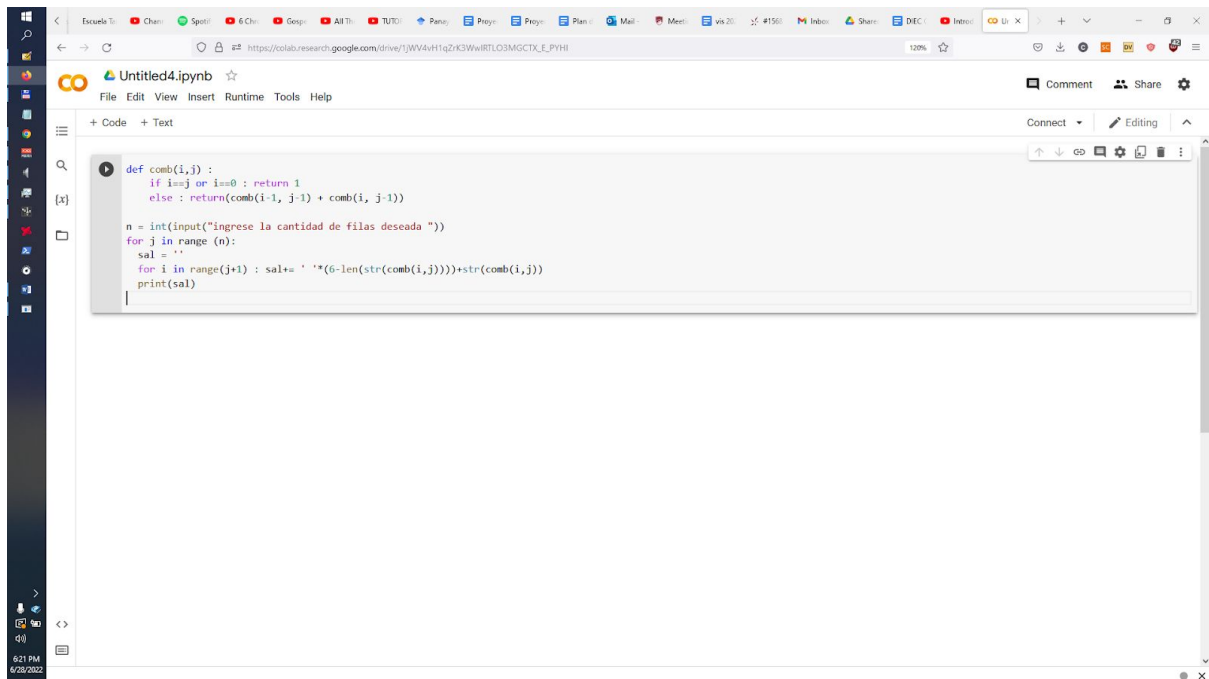
Las NB que trabajemos desde nuestra cuenta van a quedar registradas y actualizadas automáticamente en nuestro Google Drive en una carpeta específica que el mismo servidor va a crear.

En mi caso, ingresando a tal URL la pantalla a la que accedo es

Si quiero generar un nuevo “proyecto” entonces voy a New Notebook y me genera un entorno nuevo. Ver el título automático que se genera, y la “celda” donde podemos insertar código:



En esa celda podemos copypastear cualquier código python que queramos. Ver que el editor es inteligente y pinta cada parte del código de acuerdo a su función:



The screenshot shows a web browser window with a Jupyter Notebook interface. The browser's address bar displays a Google Drive link. The notebook's title bar reads "Untitled4.ipynb". The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with icons for code and text cells. The main area contains a single code cell with the following Python code, which is syntax-highlighted: a recursive function for combinations, an input prompt, and a loop that prints combinations with a specific formatting. The code is as follows:

```
def comb(i,j) :  
    if i==j or i==0 : return 1  
    else : return(comb(i-1, j-1) + comb(i, j-1))  
  
n = int(input("ingrese la cantidad de filas deseada "))  
for j in range (n):  
    sal = ''  
    for i in range(j+1) : sal+= ' '* (6-len(str(comb(i,j))))+str(comb(i,j))  
    print(sal)
```


Dándole al ícono de play se ejecuta el código. Observar cómo al ejecutar se inicializa un entorno virtual y nos muestra los indicadores de RAM y “disco”.

The screenshot shows a Google Colab notebook titled "Untitled4.ipynb". The code cell contains a function `comb(i,j)` that calculates binomial coefficients. It prompts the user to enter the number of rows, which is 16. The output displays a Pascal's triangle with 16 rows, where each row contains binomial coefficients. The status bar at the bottom indicates the execution completed at 6:33 PM.

```
def comb(i,j):
    if i==j or i==0: return 1
    else: return(comb(i-1, j-1) + comb(i, j-1))

n = int(input("ingrese la cantidad de filas deseada "))
for j in range(n):
    sal = ''
    for i in range(j+1): sal+= ' '*(6-len(str(comb(i,j))))+str(comb(i,j))
    print(sal)
```

ingrese la cantidad de filas deseada 16

```
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
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 14 1
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1
```

Como todo otro documento de Google, lo puedo compartir, y varias personas que yo autorice pueden acceder a programar en esta NB.

Presionando en +code puedo abrir otra celda (que se articula con las anteriores compar- tiendo el entorno de referenciamiento) y con +text puedo escribir texto.

The screenshot shows the same Google Colab notebook with a new code cell added. The code cell contains the same `comb` function and execution logic. Below it, a text cell contains the output of `print(n)`, which is 16. The status bar at the bottom indicates the execution completed at 6:39 PM.

```
def comb(i,j):
    if i==j or i==0: return 1
    else: return(comb(i-1, j-1) + comb(i, j-1))

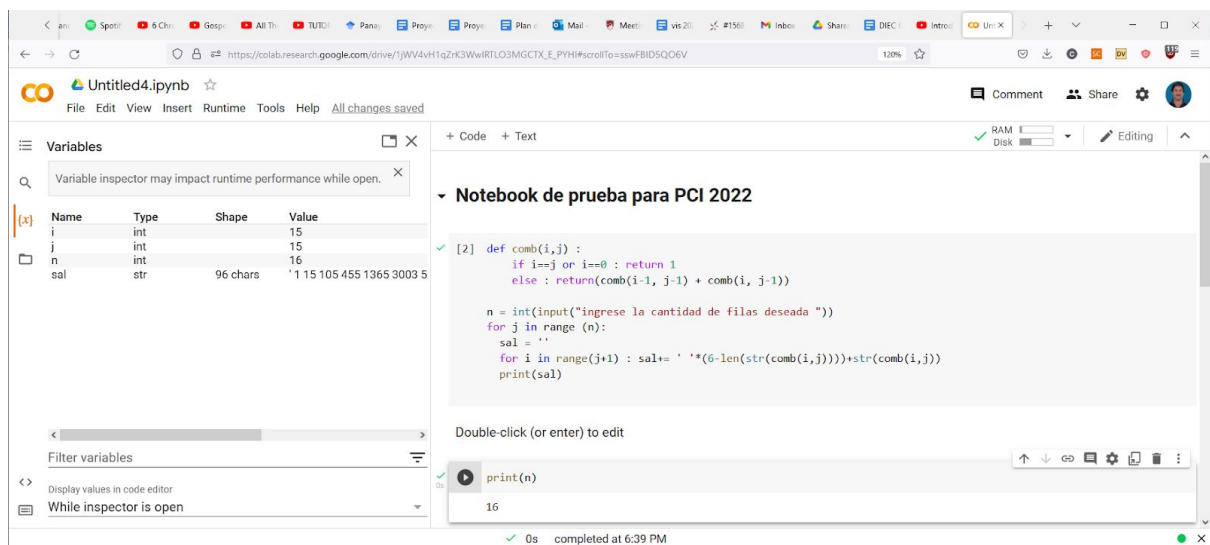
n = int(input("ingrese la cantidad de filas deseada "))
for j in range(n):
    sal = ''
    for i in range(j+1): sal+= ' '*(6-len(str(comb(i,j))))+str(comb(i,j))
    print(sal)
```

Double-click (or enter) to edit

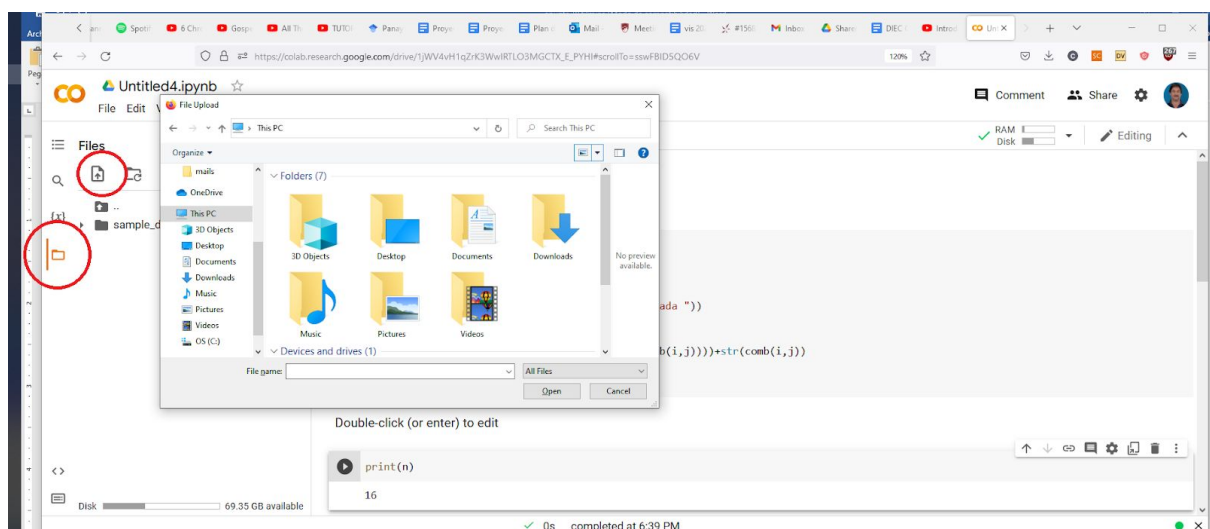
```
print(n)
```

16

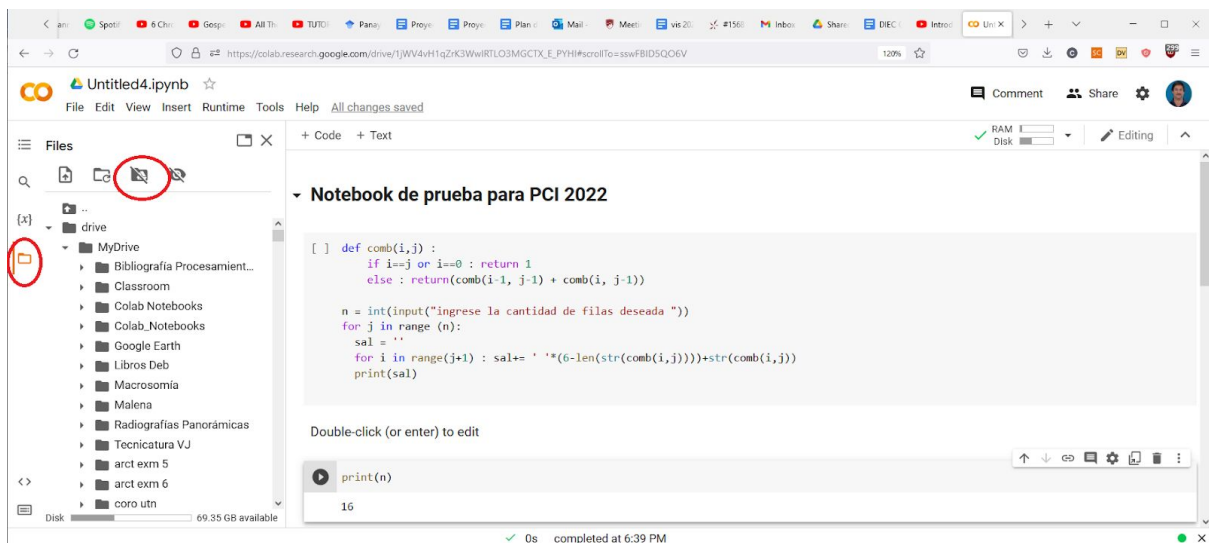
Me puedo dar cuenta cuáles son las referencias activas con el inspector, cuya ventana se abre con el símbolo x:



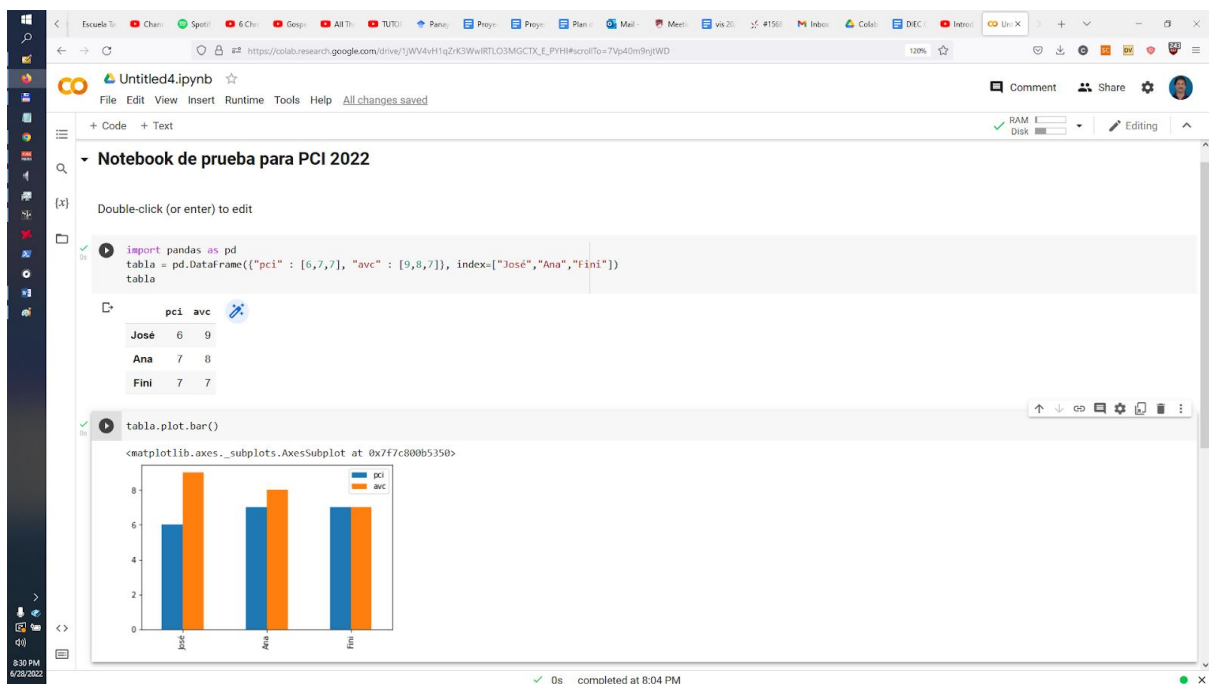
Además de escribir o copypastear código en las celdas, es posible también importarlo desde archivos. Esto se puede hacer o bien desde el disco local desde donde se ejecuta la NB, o bien desde el Google Drive del “dueño” de la NB u otro usuario que nos haya dado acceso, o bien generando un link desde el archivo y copiándolo en el NB.



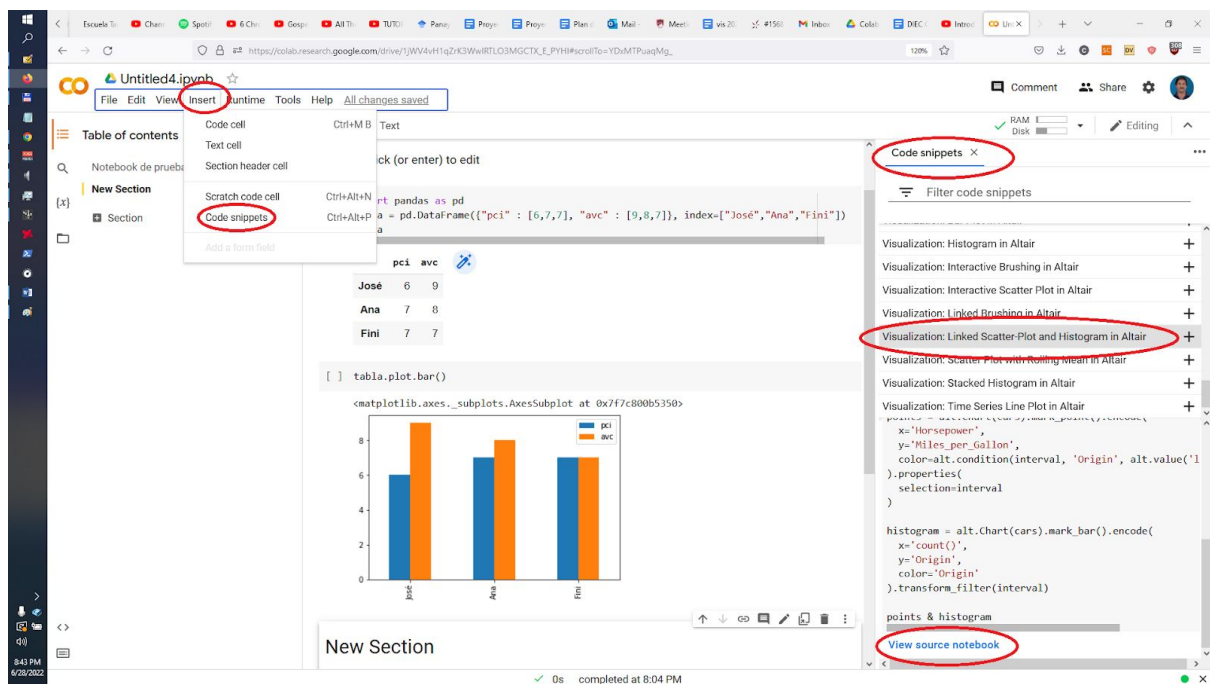
Acceso al árbol de directorios de mi drive:



Un aspecto muy importante de colab es que ya tiene “precargadas” la mayoría de las bibliotecas que puedan llegar a necesitarse.



Ya vienen incluidos algunos ejemplos de código (y sus bibliotecas requeridas) para varias funciones.



The screenshot shows a Jupyter Notebook titled 'Untitled4.ipynb'. The 'Insert' menu is open, and 'Code snippets' is highlighted. The code cell contains the following Python code:

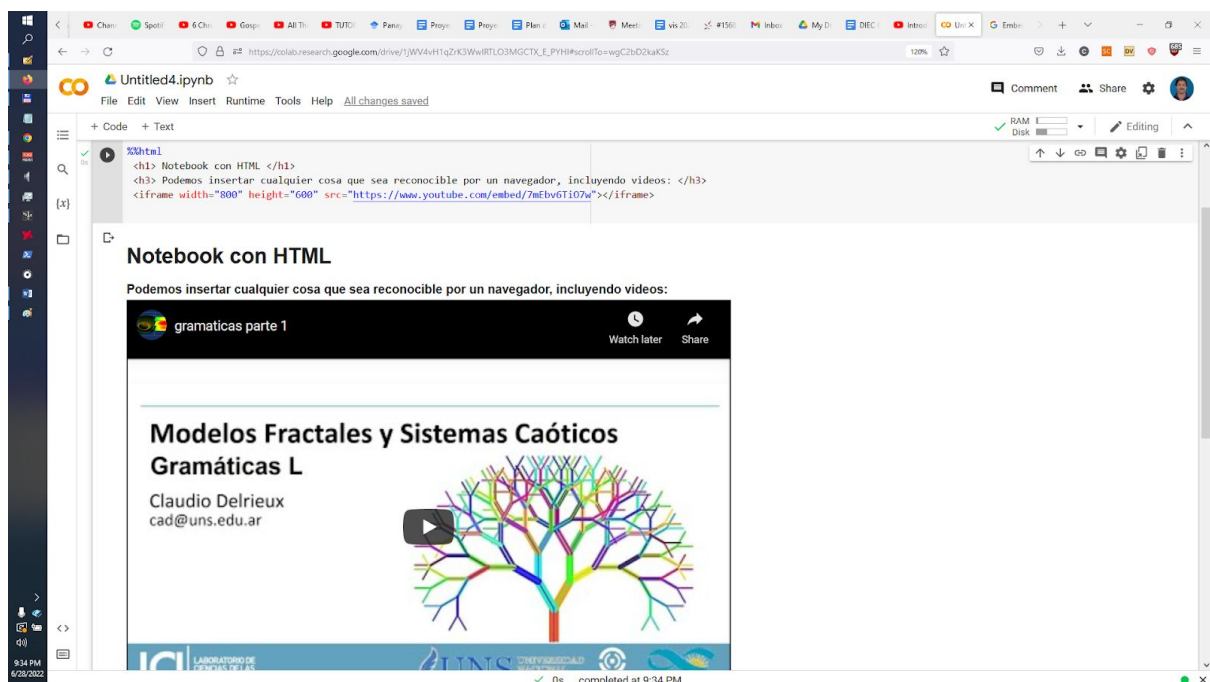
```
import pandas as pd
a = pd.DataFrame({"pci" : [6,7,7], "avc" : [9,8,7]}, index=["José", "Ana", "Fini"])

pci avc
José 6 9
Ana 7 8
Fini 7 7
```

Below the code is a bar chart titled 'tabla.plot.bar()'. The chart has two series: 'pci' (blue bars) and 'avc' (orange bars). The x-axis labels are 'José', 'Ana', and 'Fini'. The y-axis ranges from 0 to 8. The 'avc' series has values 9, 8, and 7 for the respective categories.

On the right side, there is a 'Code snippets' sidebar. It lists various Altair visualizations, including 'Visualization: Histogram in Altair', 'Visualization: Interactive Brushing in Altair', 'Visualization: Interactive Scatter Plot in Altair', 'Visualization: Linked Scatter Plot and Histogram in Altair' (which is selected), 'Visualization: Scatter Plot with Rolling Mean in Altair', 'Visualization: Stacked Histogram in Altair', and 'Visualization: Time Series Line Plot in Altair'. Below the list, there is a 'View source notebook' link.

Es también posible insertar código HTML en una celda, poniendo previamente un flag:



The screenshot shows a Jupyter Notebook titled 'Untitled4.ipynb'. The code cell contains the following HTML code:

```
<html>
<h1> Notebook con HTML </h1>
<h3> Podemos insertar cualquier cosa que sea reconocible por un navegador, incluyendo videos: </h3>
<iframe width="800" height="600" src="https://www.youtube.com/embed/7mEb6t107w" ></iframe>
```

The output of the code cell shows a video player. The video is titled 'Modelos Fractales y Sistemas Caóticos Gramáticas L' by Claudio Delrieux, with the email address 'cad@uns.edu.ar'. The video player includes a play button and a 'Watch later' button. The video content shows a fractal tree structure.