



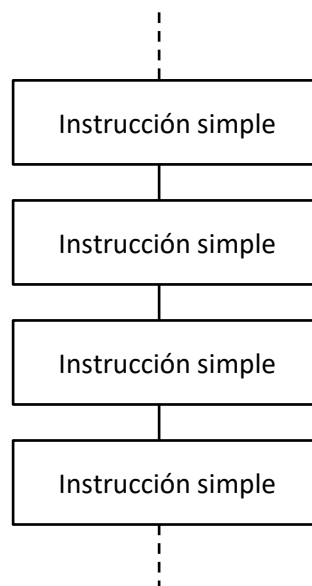
# Ficha 2

## Estructuras Secuenciales

### 1.] Resolución de problemas simples. Estructuras secuenciales.

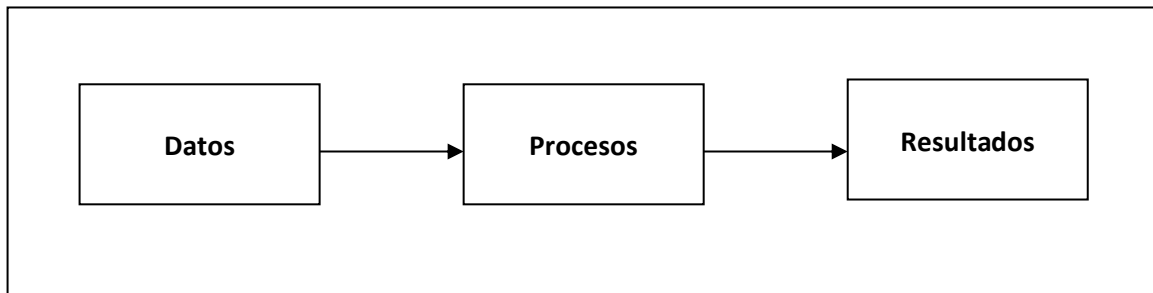
Con lo visto hasta ahora estamos en condiciones de analizar *problemas simples* de forma de poder desarrollar programas que los resuelvan. Como veremos en una ficha posterior, un problema se dice *simple* si no admite (o no justifica) ser dividido en problemas menores o *subproblemas* [1]. Por ahora, nos concentraremos en problemas simples de *lógica lineal* que pueden ser resueltos mediante la aplicación de *secuencias de instrucciones simples* (recuerde que básicamente, una *instrucción simple* es una asignación, o una instrucción de visualización o una instrucción de carga por teclado) una debajo de la otra, de tal manera que cada instrucción se ejecuta una después de la otra. Un bloque de instrucciones lineal de esta forma, se conoce en programación como un *bloque secuencial de instrucciones* o también como una *estructura secuencial de instrucciones*. La *Figura 1* muestra un esquema general de una estructura secuencial:

Figura 1: Esquema general de una *Estructura Secuencial* de instrucciones.



Como vimos en la Ficha 01, antes de escribir un programa debemos ser capaces de plantear un *algoritmo* que muestre la forma de resolver el problema que se analiza. Recordemos que un *algoritmo* es la secuencia de pasos que permite resolver un problema, y por lo tanto su planteo es el paso previo a un *programa*. En ese sentido, *resolver algorítmicamente un problema* significa plantear el mismo de forma tal que queden indicados los pasos necesarios para obtener los resultados pedidos a partir de los datos conocidos. Y como también vimos en la Ficha 01, esto implica que en un algoritmo básicamente intervienen tres elementos: *datos*, *procesos* y *resultados* (ver *Figura 2*):

Figura 2: Estructura de un Algoritmo



Dado un problema, entonces, para plantear un algoritmo que permita resolverlo es siempre conveniente *entender correctamente el enunciado* y tratar de deducir y caracterizar a partir del mismo los elementos ya indicados (datos, procesos y resultados). Lo normal es:

1. Comenzar identificando los **resultados** pedidos, porque así quedan claros los objetivos a cumplir.
2. Luego individualizar los **datos** con que se cuenta y determinar si con estos es suficiente para llegar a los resultados pedidos.
3. Finalmente si los datos son completos y los objetivos claros, se intentan plantear los **procesos** necesarios para convertir esos datos en los resultados esperados.

Cabe aclarar que cuando se pide identificar resultados y datos, no se trata simplemente de enumerarlos sino, además, de *caracterizarlos*: esto es, poder indicar de qué *tipo* de datos y resultados se está hablando (números enteros o flotantes, cadenas, valores lógicos, etc.), que rango de valores son aceptables (¿negativos? ¿sólo positivos? ¿vale el cero? ¿pueden venir en una cadena caracteres que representen números mezclados con otros que representen letras? etc.) así como de asociar a cada uno un identificador genérico que luego se convierta en el nombre de una variable. En cuanto a los procesos, en esta fase de comprensión del problema se espera que el programador sepa al menos identificar el contexto del problema (¿incluye alguna noción matemática esencial? ¿el volumen de los datos a procesar requerirá de algoritmos sofisticados? ¿es necesario algún tipo de desarrollo gráfico?, ¿es similar a otros problemas ya analizados?, etc.) para que pueda comenzar a explorar posibles soluciones y/o interiorizarse sobre técnicas especiales que podría requerir.

Es completamente cierto que un programador experimentado lleva a cabo este proceso de comprensión del problema en forma mental, sin tanto formalismo, pero en el caso de un curso introductorio conviene, al menos en los primeros ejercicios, plantearlo en forma rigurosa para fijar las ideas y facilitar la discusión.

Una vez cumplido el paso anterior, se procede al planteo del algoritmo (o al menos, a intentarlo). En este punto, el programador buscará determinar el conjunto de instrucciones o pasos que permitan pasar del conjunto de datos al conjunto de resultados. En ese sentido, el algoritmo actúa como un proceso que *convierte* esos datos en los resultados pedidos, como una función que opera sobre las variables de entrada. El programador combina los diferentes tipos de instrucciones que sabe que son soportadas por los lenguajes de programación (es decir, el conjunto de *operaciones primitivas* válidas) y plantea el esquema de pasos a aplicar. Cuando el problema es complejo y el algoritmo es extenso y difícil de plantear, se suelen usar técnicas auxiliares que ayudan al programador a poner sus ideas en claro. Entre esas técnicas se cuentan los *diagramas de flujo* y el *pseudocódigo*, que analizaremos más adelante.



Los tres ejemplos de problemas que siguen serán resueltos aplicando *dos fases*, con el objetivo de poner en práctica los conceptos vistos hasta aquí: primero, se identificarán y caracterizarán *resultados, datos y procesos*; y luego se concretará un algoritmo para resolver el problema. Como en los tres casos el problema es de naturaleza **muy** sencilla, no se requerirá ninguna técnica auxiliar para plantear el algoritmo y por lo tanto el mismo será plasmado directamente como un *programa en Python* (esto no es incorrecto ni inconsistente, ya que como sabemos, un *programa es un algoritmo*).

Consideremos el enunciado del siguiente problema básico, como primer ejemplo:

**Problema 1.)** *Dado el valor de los tres lados de un triángulo, calcular el perímetro del triángulo.*

**a.) Identificación de componentes:** Para cada componente identificado, se asocia un nombre, que luego será usado como variable en el programa correspondiente.

- **Resultados:** El perímetro de un triángulo. (*p*: coma flotante)
- **Datos:** Los tres lados del triángulo. (*lad1, lad2, lad3*: coma flotante)
- **Procesos:** Problema de *geometría elemental*. El perímetro *p* de un triángulo es igual a la suma de los valores de sus tres lados *lad1, lad2* y *lad3*, por lo que la fórmula o expresión a aplicar es:  $p = lad1 + lad2 + lad3$ . En este caso, podemos suponer que los valores de los lados vendrán dados de forma que efectivamente puedan conformar un triángulo, pero en una situación más realista el programador debería *validar* (es decir, controlar y eventualmente volver a cargar) los valores ingresados en *lad1, lad2* y *lad3* de forma que al menos se cumplan las relaciones o propiedades elementales de los lados de un triángulo: *la longitud de cualquiera de los lados de un triángulo es siempre menor que la suma de los valores de los otros dos, y mayor que su diferencia*.

**b.) Planteo del algoritmo (como script o programa en Python):** Más adelante en esta misma Ficha (sección 6, página 46) se muestra la forma de usar el entorno de programación *PyCharm* para Python. Puede leer esa sección antes de continuar, si lo cree necesario. Asegúrese de poder crear un proyecto nuevo en *PyCharm*, abrir el editor de textos y escribir el siguiente programa para poder probarlo y modificarlo si lo cree conveniente:

```
__author__ = 'Cátedra de AED'

# título general y carga de datos...
print('Ejemplo 1 - Perímetro de un triángulo')
lad1 = float(input('Longitud del primer lado: '))
lad2 = float(input('Longitud del segundo lado: '))
lad3 = float(input('Longitud del tercer lado: '))

# procesos...
p = lad1 + lad2 + lad3

# visualización de resultados...
print('El perímetro es:', p)
```

En cuanto a la *lógica general*, el programa anterior es el reflejo directo del análisis de procesos que se hizo en el paso anterior. Se cargan por teclado los valores de los tres lados (sin lo cual no se puede hacer la suma). Luego se aplica la fórmula del perímetro que se puso en claro en el mismo análisis del proceso y el resultado *se asigna* en la variable *p*. Y finalmente se muestra el resultado final en la consola de salida. Note que las variables usadas se designaron con los



mismos nombres que los que se usaron al identificar resultados y datos, y que en el planteo del script se respetaron los *tipos de datos* generales que fueron caracterizados en esa misma fase. Como dijimos, en una situación realista el programador debería incluir algún tipo de control para validar que los lados que se cargan cumplan con las propiedades que se indicaron en el análisis de procesos, pero en este ejemplo introductorio podemos obviar ese detalle y asumir que los datos serán cargados correctamente (de hecho, también se debería validar que los valores de los lados no sean negativos ni cero).

En cuanto a detalles de estilo, observe que se mantuvo en el script la variable `__author__` asignada con el nombre del autor del programa (que *PyCharm* inserta por default, aunque eso depende de la versión de *PyCharm* que esté usando). También note el uso de líneas de *comentarios de texto* en distintos lugares del script para hacer más clara su lectura y su comprensión, así como el uso de líneas en blanco para separar bloques de instrucciones afines (aportando también claridad). El script arranca mostrando un título básico que hace más amigable y comprensible su uso por parte del usuario. **Y por supuesto, todo el programa está correctamente indentado (encolumnado): no podría ejecutarse si no fuese así.**

Veamos ahora el segundo ejemplo:

**Problema 2.)** *Se conoce la cantidad de horas que trabaja un empleado en una fábrica, más el importe que percibe por cada hora trabajada, además del nombre del empleado. Se pide calcular el importe final del sueldo que el empleado deberá cobrar y mostrar el nombre del empleado y el importe final del sueldo que se calculó.*

**a.) Identificación de componentes:**

- **Resultados:** Nombre del empleado. (*nom*: cadena de caracteres)  
Sueldo final. (*sueldo*: número en coma flotante)
- **Datos:** Nombre del empleado. (*nom*: cadena de caracteres)  
Horas trabajadas. (*horas*: número entero)  
Monto ganado por hora. (*monto*: número en coma flotante)
- **Procesos:** Problema de *gestión administrativa/contable*. El nombre del empleado debe ser informado en la consola de salida tal como se cargue en la variable *nom*. En cuanto al sueldo final (variable *sueldo*), se deduce claramente la operación a realizar: multiplicar la cantidad de horas trabajadas (variable *horas*) por el importe que se le paga al empleado por cada hora trabajada (variable *monto*) Por lo tanto, la fórmula o expresión será:  $sueldo = horas * monto$ . De nuevo, en una situación real el programador debería validar que los valores cargados en *horas* y *monto* sean *mayores a cero*, pero por ahora no nos preocuparemos por ese detalle: asumiremos que el usuario cargará correctamente ambos datos.

**b.) Planteo del algoritmo (como script o programa en *Python*):** Asegúrese de poder agregar este programa como parte del *mismo proyecto que creó para el Problema 1*... ¡No necesita crear un nuevo proyecto para cada programa o script que desarrolle!

```
__author__ = 'Cátedra de AED'
```

```
# título general y carga de datos...
print('Ejemplo 2 - Cálculo del sueldo de un empleado')
nom = input('Ingrese el nombre del empleado: ')
horas = int(input('Ingrese la cantidad de horas trabajadas: '))
```



```
monto = float(input('Ingrese el monto a cobrar por hora: '))

# procesos...
sueldo = horas * monto

# visualizaci n de resultados...
print('Empleado: ', nom, '- Sueldo a cobrar:', sueldo, 'pesos')
```

En este script nuevamente tenemos una traducci n directa a Python del an lisis de procesos que se hizo en el paso anterior. El enunciado no dice formalmente que el sueldo final es el resultado de la multiplicaci n entre las horas trabajadas y el monto a pagar por hora, pero se deduce f cilmente del contexto. Por otra parte, no es extra o que se solicite mostrar en la pantalla de salida alg n dato original (como el nombre) sin cambios. Otra vez, note el uso de comentarios de texto, l neas en blanco separadoras, la visualizaci n de un t tulo general, el uso de mensajes claros al cargar datos y al mostrar los resultados, y la **correcta indentaci n del script para evitar errores de int rprete**.

Nuestro tercer ejemplo es el que sigue:

**Problema 3.)** *Se tiene registrada la temperatura ambiente medida en tres momentos diferentes en un dep sito de qu micos y se necesita calcular el valor promedio entre las temperaturas medidas, tanto en formato entero (sin decimales) como en formato real (con decimales).*

**a.) Identificaci n de componentes:**

- **Resultados:** Promedio entero. (*prom1*: n mero entero)  
Promedio real. (*prom2*: n mero en coma flotante)
- **Datos:** Tres temperaturas. (*t1, t2, t3*: n meros enteros)
- **Procesos:** Problema de *aritm tica elemental*. El enunciado no especifica si las tres temperaturas ser n n meros enteros o de coma flotante, por lo que asumimos que ser n enteros. El c lculo del promedio es simplemente la suma de las tres temperaturas *t1, t2* y *t3*, dividido por 3. Como se piden dos promedios, podemos almacenar la suma en una variable auxiliar ( $suma = t1 + t2 + t3$ ) y luego dividir *suma* por 3. El cociente entero puede calcularse en Python con el operador `//` (y entonces ser a  $prom1 = suma // 3$ ) mientras que el cociente real (o de coma flotante) puede calcularse con el operador `/` (o sea,  $prom2 = suma / 3$ ). Note que en este problema, los valores de las temperaturas podr an ser negativos, cero o positivos, por lo que incluso en una situaci n real no habr a tanto problema con la validaci n de los datos que se carguen.

**b.) Planteo del algoritmo (como script o programa en Python):** Agregue este programa como parte del mismo proyecto que cre  para el *Problema 1*:

```
__author__ = 'C tedra de AED'

# t tulo general y carga de datos...
print('Ejemplo 3 - C lculo de la temperatura promedio')
t1 = int(input('Temperatura 1: '))
t2 = int(input('Temperatura 2: '))
t3 = int(input('Temperatura 3: '))

# procesos...
suma = t1 + t2 + t3
```



```
prom1 = suma // 3
prom2 = suma / 3

# visualización de resultados...
print('Promedio entero:', prom1, '- Promedio real:', prom2)
```

Este script sigue siendo elemental, pero es un poco más extenso que los dos ejemplos anteriores. Aquí, la secuencia de instrucciones que constituye el proceso de los datos, incluye tres operaciones: la suma y los dos cocientes. Otra vez, note el uso de comentarios de texto, líneas en blanco separadoras, la visualización de un título general, el uso de mensajes claros al cargar datos y al mostrar los resultados, y la **correcta indentación del script para evitar errores de intérprete**.

Observe cómo el uso de la variable auxiliar *suma* (no necesariamente prevista en el paso de identificación de componentes) ayuda a un planteo más claro y sin redundancia de operaciones: sin esa variable, el programador debería calcular la suma dos veces para calcular los dos promedios. Por otra parte, a modo de adelanto del tema que veremos en la próxima sección, consideremos solamente el cálculo del promedio real entre los valores de las variables *t1*, *t2* y *t3* pero sin el uso de la variable auxiliar. El promedio es la suma de los tres valores, dividido por 3 y en principio el cálculo en Python podría escribirse (*ingenuamente...*) así:

```
p = t1 + t2 + t3 / 3
```

**Lo anterior es un error:** así planteada la expresión, lo que se divide por 3 es *sólo el valor de la variable t3* y **no** la suma de las tres variables. Así escrita, la asignación anterior es equivalente a:

```
p = t1 + t2 + (t3 / 3)
```

que de ninguna manera calcula el promedio pedido. La forma correcta de escribir la instrucción, debió ser:

```
p = (t1 + t2 + t3) / 3
```

## 2.] Técnicas de representación de algoritmos: Diagramas de flujo.

Hemos indicado que el paso previo para el desarrollo de un programa es el planteo de un algoritmo que muestre la forma de resolver el problema que se está enfrentando. Los problemas que hemos analizado eran lo suficientemente simples como para que el algoritmo fuese directo: unas pocas consideraciones respecto del proceso a realizar y luego se pudo escribir el programa sin dificultades adicionales.

Sin embargo, en la gran mayoría de los casos, los problemas o requerimientos que enfrenta un programador no son tan sencillos ni tan directos. La lógica detrás del algoritmo puede ser compleja, con demasiadas ramas lógicas derivadas del uso de condiciones (que veremos) o demasiado cargada con situaciones especiales. Incluso un programador experimentado podría tener dificultades para captar la estructura de esa lógica en casos complejos, o bien, podría querer transmitir aspectos del algoritmo a otros programadores sin entrar en demasiado detalle de complejidad, y en este caso el idioma hablado resulta ambiguo.

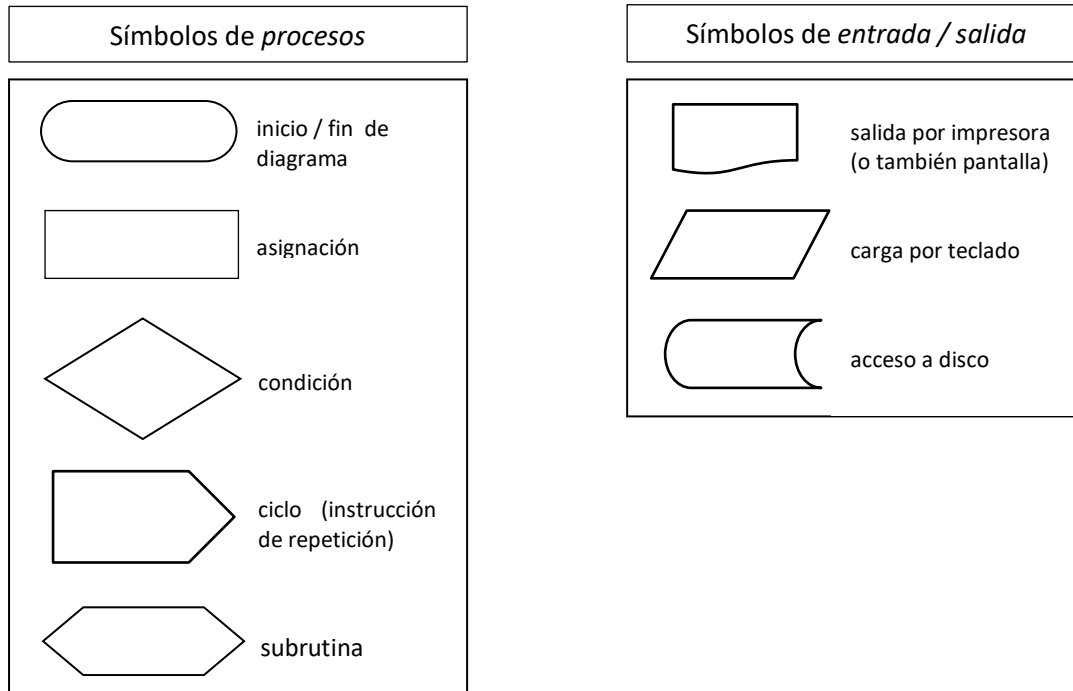
En casos así suele ser útil contar con herramientas de representación general de algoritmos, tales como gráficos, que permitan que un programador pueda rescatar y reflejar con más claridad la lógica general del algoritmo, sin tener que preocuparse (por ejemplo) de los elementos de sintaxis específica de un lenguaje [1].

Los *diagramas de flujo* y el *pseudocódigo* son dos de estas técnicas que ayudan a hacer unívoca la representación del algoritmo. Cada una de ambas, a su vez, admite innumerables variantes que no deben preocupar demasiado al estudiante: siempre recuerde que se trata de *técnicas auxiliares* en el trabajo de un programador, que normalmente serán usadas a modo de borrador o ayuda circunstancial, y podrá adaptarlas a sus preferencias (salvo en casos en que el programador trabaje en un contexto en el cual se hayan acordado reglas especiales para el planteo de un diagrama o de un pseudocódigo, y aun así, esas reglas no significarán un problema).

Un *diagrama de flujo* es un gráfico que permite representar en forma clara y directa el algoritmo para resolver un problema. Se basa en el uso de unos pocos símbolos unidos por líneas rectas descendentes. Los símbolos de un *diagrama de flujo* pueden clasificarse en dos grupos, y los más comunes de ellos se pueden ver en la *Figura 3*:

- ✓ **Símbolos de representación de procesos:** cada uno representa una operación para transformar datos en resultados. A modo de ejemplo, el símbolo usado para representar instrucciones de asignación es un rectángulo y forma parte de este grupo.
- ✓ **Símbolos de representación de operaciones de entrada y/o salida:** cada símbolo representa una acción o instrucción mediante la cual se lleva a cabo alguna operación de carga de datos o salida de resultados a través de algún dispositivo. En nuestros diagramas, se usarán para representar instrucciones de carga o entradas por teclado (mediante la función `input()` de Python 3) y también para las invocaciones a la función de salida por consola estándar `print()` de Python 3.

**Figura 3: Principales símbolos usados en un diagrama de flujo.**



El símbolo que hemos indicado como *carga por teclado* originalmente se usa tanto para indicar una *entrada genérica* como una *salida genérica*, en forma indistinta. Aquí, *genérica* significa que no es necesario en ese momento indicar en forma precisa el tipo de dispositivo a usar y sólo importa rescatar que se espera una operación de entrada o de salida. Sin embargo, para



evitar ambigüedad, aquí usaremos este símbolo *sólo para indicar una entrada* (no una salida), normalmente desde el teclado.

Y en el mismo orden de cosas, el símbolo que hemos indicado como *salida por impresora* originalmente sólo se usa para eso. Pero en este contexto lo admitiremos también para indicar una salida por pantalla o consola estándar. Con este par de convenciones, tratamos de evitar confusiones en cuanto al uso de símbolos que representen carga o visualización en un diagrama, pero no lo olvide: un diagrama de flujo es una herramienta auxiliar y adaptable a las necesidades del programador. Si se trabaja en un contexto colaborativo, simplemente deben pactarse las convenciones de representación y facilitar así el trabajo del equipo [1].

A modo de ejemplo de aplicación, considere el siguiente problema simple:

**Problema 4.)** *Se conoce la cantidad total de personas que padecen cierta enfermedad en todo el país, y también se sabe cuántas de esas personas viven en una ciudad determinada. Se desea saber qué porcentaje representan estas últimas sobre el total de enfermos del país.*

**a.) Identificación de componentes:**

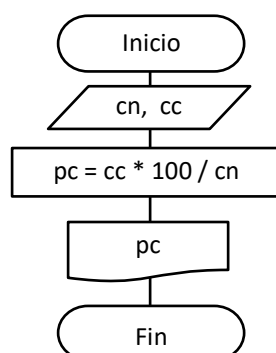
- **Resultados:** Porcentaje de enfermos de la ciudad con respecto al total nacional.  
(*pc*: número de coma flotante (o *real*))
- **Datos:** Cantidad de enfermos (nacional). (*cn*: número entero)  
Cantidad de enfermos (ciudad). (*cc*: número entero)
- **Procesos:** Problema de *cálculo de porcentaje en contexto de salud humana*. El cálculo del porcentaje surge de una regla de tres: La cantidad de enfermos de todo país (*cn*) equivale al 100% del total y la cantidad de enfermos en la ciudad (*cc*) equivale al porcentaje pedido *pc*, que entonces sale de:  $pc = cc * 100 / cn$ .

La expresión anterior para el cálculo del porcentaje no incluye sumas ni restas, y los dos operadores que aparecen (*producto* y *división real*) son de la misma precedencia, por lo que primero se aplicará el producto, y luego la división. Pero en términos del resultado final, sería lo mismo dividir primero y multiplicar después, lo que podría expresarse así:  $pc = cc * (100 / cn)$ .

Note que en una situación real, el programador debería incluir algún tipo de validación al cargar los valores *cn* y *cc*, para evitar que se ingresen negativos.

**b.) Planteo del algoritmo:** Si bien este problema es tan sencillo que sólo con lo anterior ya bastaría para pasar directamente al programa, mostramos aquí el algoritmo en forma de *diagrama de flujo* para comenzar a ejercitar con esa herramienta:

**Figura 4: Diagrama de flujo del problema de cálculo del porcentaje.**







Como se ve, el diagrama se construye y se lee *de arriba hacia abajo*. Los símbolos que lo componen se unen entre sí mediante *líneas rectas verticales*, y como la lectura es descendente, se asume que esas líneas de unión también lo son: no es necesario hacer que terminen en punta de flecha (aunque podría hacerse si el programador lo desea).

El diagrama comienza y termina con el símbolo de *inicio / fin de diagrama* (ver *Figura 3*). En el inicial se escribe la palabra *Inicio* (o alguna equivalente) y en el de cierre se escribe la palabra *Fin* (o equivalente). Para indicar que se espera la carga por teclado de los valores de las variables *cn* y *cc* se usa el paralelogramo indicado como símbolo de carga por teclado (ver *Figura 3*), escribiendo dentro de él los nombres de ambas variables. El propio cálculo del porcentaje se escribe dentro del rectángulo que representa una asignación. Y el *símbolo de salida por impresora o pantalla* se usa para indicar que se espera mostrar el valor final de la variable cuyo nombre se inscribe dentro de él (en este caso, la variable *pc*).

Note que un diagrama de flujo *es genérico*: se plantea de forma que (en lo posible) no se haga referencia a lenguaje de programación alguno. La idea es que una vez planteado el diagrama, el programador escriba el programa usando el lenguaje que quiera. No hay (no debería haber...) diagramas de flujo para Python ni diagramas de flujo para Java o Basic o Pascal... *Un único diagrama debe servir de base para un programa en cualquier lenguaje*<sup>1</sup>.

En general, un diagrama de flujo *no debe incluir indicaciones de visualización de mensajes en pantalla*, salvo que eso de alguna manera ayude a entender la lógica general. Por caso, note que en el diagrama anterior se indica la carga por teclado las variables *cn* y *cc*, pero no se incluyen los mensajes que acompañan a la carga (como "*Ingrese el total nacional: "*" o "*Ingrese el total de la ciudad: "*"). Lo mismo vale en la visualización del resultado final *pc*: sólo se indica el nombre de la variable a mostrar, y se omiten los mensajes auxiliares [1].

- c.) **Desarrollo del programa:** Si se tiene un diagrama de flujo bien estructurado, el script o programa se deduce en forma prácticamente directa:

```
__author__ = 'Cátedra de AED'

# título general y carga de datos...
print('Ejemplo 4 - Cálculo del porcentaje de enfermos')
cn = int(input('Cantidad de enfermos en el país: '))
cc = int(input('Cantidad de enfermos en la ciudad: '))

# procesos...
pc = cc * 100 / cn

# visualización de resultados...
print('Porcentaje de la ciudad sobre el total país:', pc, '%')
```

### 3.] Técnicas de representación de algoritmos: Pseudocódigo.

Otra técnica muy usada por los programadores para representar genéricamente un algoritmo, es la que se conoce como *pseudocódigo*. Se trata de una herramienta por medio de la cual se plantea el algoritmo sin usar gráficas, escribiendo cada acción o paso en *lenguaje natural o cotidiano*, sin entrar en detalles sintácticos de ningún lenguaje aunque respetando ciertas

---

<sup>1</sup> No obstante, los programadores pueden tomarse algunas licencias y relajar esta regla. Si se sabe a ciencia cierta que el lenguaje será Python, por ejemplo, entonces el diagrama de flujo podría incluir instrucciones con sintaxis específica de Python (como  $y = x ** 2$ ) y nadie debería molestarse o sorprenderse por ello.



pautas básicas (como la indentación). A diferencia de un programa terminado, el *pseudocódigo* está pensado para ser leído y entendido por una persona, y no por un computador.

Dado que el programa fuente terminado en un lenguaje específico se conoce como el *código* fuente, la palabra *pseudocódigo* hace referencia a una forma de *código incompleta o informal* (o sea, un código que no llega a ser completo ni riguroso).

El nivel de profundidad que se refleje en el *pseudocódigo* depende de la necesidad del programador, así como el nivel de formalismo o estructuración en el planteo de cada paso. A modo de ejemplo, el algoritmo para el mismo *problema 4* de la sección anterior (que ya resolvimos con el diagrama de flujo de la *Figura 4*) podría replantearse mediante el siguiente esquema de *pseudocódigo*:

Algoritmo:

1. Cargar en *cn* la cantidad de enfermos del país
2. Cargar en *cc* la cantidad de enfermos de la ciudad
3. Calcular y asignar en *pc* el porcentaje:  $pc = cc * 100 / cn$
4. Mostrar el porcentaje *pc*

Está claro que tratándose de un problema tan sencillo, es posible que no sea necesaria tanta minuciosidad o detalle en la descripción informal de cada paso. Si se desea claridad en el significado de cada variable o cálculo, lo anterior es aceptable. Pero en muchos casos, el siguiente replanteo más reducido puede ser suficiente (si los programadores conocen el contexto del problema):

Algoritmo:

1. Cargar *cn*
2. Cargar *cc*
3. Sea  $pc = cc * 100 / cn$
4. Mostrar *pc*

Como se ve, quien plantea el pseudocódigo dispone de mucha libertad para expresar cada paso. *No hay un estándar general* y las convenciones y reglas de trabajo pueden variar mucho entre programador y programador, pero en muchos ámbitos se suelen aceptar las siguientes (que son las que básicamente emplearemos en este curso cuando se requiera el planteo de pseudocódigo)<sup>2</sup>:

- ✓ Comenzar indicando en la primera línea el *nombre del proceso* que se está describiendo, seguido de *dos puntos* (como veremos, esto es típico de Python). Si no se conoce el nombre del proceso, escriba un descriptor genérico (*Algoritmo* en nuestro caso).
- ✓ Enumerar cada paso en forma correlativa. Si hubiese *subpasos*, enumerarlos en base al paso principal<sup>3</sup>:

---

<sup>2</sup> Obviamente, existen muchas otras convenciones y reglas y tanto el estudiante como sus profesores pueden aplicar las que deseen.

<sup>3</sup> Veremos en una Ficha posterior que una instrucción puede estar compuesta por otras instrucciones incluidas en ella, lo cual la hace una *instrucción compuesta*. En contrapartida, una *instrucción simple* (como una asignación o una visualización o una carga) no contiene instrucciones incluidas.

Algoritmo:

1. Paso 1...
2. Paso 2...
  - 2.1. Subpaso 2.1...
  - 2.2. Subpaso 2.2...
3. Paso 3...

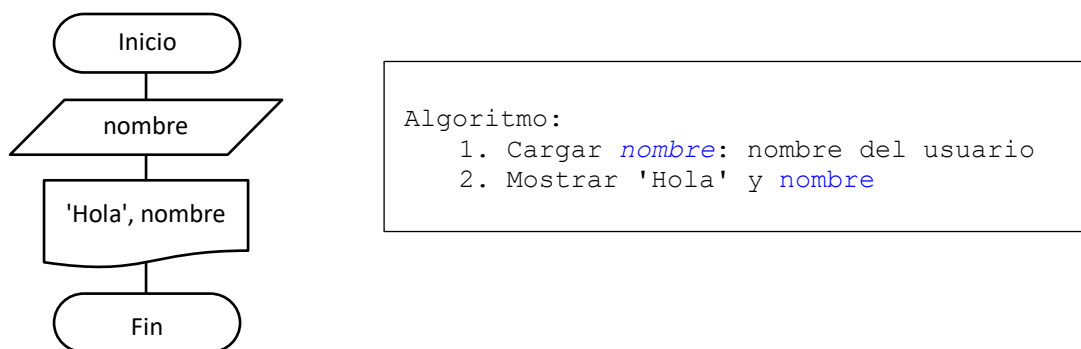
- ✓ Mantener el encolumnado o indentación general, tanto a nivel de pasos principales como a nivel de *subpasos* (ver ejemplo anterior).
- ✓ Mantener consistencia: si tiene (por ejemplo) dos pasos de carga, designe a ambos de la misma forma y con el mismo estilo. El siguiente replanteo del pseudocódigo que vimos, *no respeta la consistencia*:

1. Cargar la variable *cn*
2. Ingrese en *cc* un valor
3. Sea  $pc = cc * 100 / cn$
4. Mostrar *pc*

Como dijimos, cada programador decide la forma y el nivel de detalle y profundidad de su pseudocódigo. En muchos casos, se usa el criterio de plantear pseudocódigos usando una estructura análoga a la del lenguaje que se sabe que se usará luego para el desarrollo del programa. En general, mientras más se basa un pseudocódigo en las reglas y convenciones sintácticas de un lenguaje en particular, *más estructurado se dice que es ese pseudocódigo*. En nuestro caso, a lo largo del curso emplearemos pseudocódigo con bastante libertad, aunque siguiendo una estructura básica inspirada en Python.

Como ya se dijo, tanto los diagramas de flujo como el pseudocódigo son técnicas auxiliares de representación de algoritmos. En teoría, al plantear un diagrama o un pseudocódigo se hace de forma genérica, tratando de no apegarse a sintaxis ni convenciones de ningún lenguaje, aunque en la práctica esto se relaja, dado que los programadores saben en qué lenguaje trabajarán. Pero aun así, un mismo diagrama y/o un mismo pseudocódigo debe plantearse de forma que pueda servir como base para un programa en cualquier lenguaje. A modo de ejemplo, si se quiere escribir un programa que cargue por teclado el nombre de una persona y a continuación le muestre un saludo por consola estándar, podría tener la siguiente estructura algorítmica elemental planteada como diagrama y como pseudocódigo:

Figura 5: Diagrama y pseudocódigo para el problema de cargar el nombre y mostrar un saludo



El diagrama y el pseudocódigo que se ven arriba, sirven sin problemas para guiar el desarrollo de un programa en el lenguaje que prefiera el programador. Como ejemplo, los siguientes son



cuatro programas en cuatro lenguajes diferentes (*Pascal*, *Java*, *C++* y *Python*), que implementan el algoritmo reflejado en el diagrama y el pseudocódigo [2]:

**Figura 6: El mismo algoritmo en cuatro programas con diferentes lenguajes.**

<p><b>Pascal:</b></p> <pre>Program Hola; Uses CRT; Var     nombre: String[40]; Begin     WriteLn("Ingrese su nombre: ");     ReadLn(nombre);     WriteLn("Hola ", nombre); End.</pre>	<p><b>Java:</b></p> <pre>import java.util.Scanner; public class Hola {     public static void main(String [] args)     {         Scanner sc = new Scanner(System.in);          String nombre;          System.out.println("Ingrese su nombre: ");         nombre = sc.nextLine();         System.out.println("Hola " + nombre);     } }</pre>
<p><b>C++:</b></p> <pre>#include&lt;iostream&gt; using namespace std; int main() {     char nombre[40];      cout &lt;&lt; "Ingrese su nombre: ";     cin &gt;&gt; nombre;     cout &lt;&lt; "Hola " &lt;&lt; nombre &lt;&lt; endl;     return 0; }</pre>	<p><b>Python:</b></p> <pre>nombre = input("Ingrese su nombre: ") print("Hola", nombre)</pre>

#### 4.] Precedencia de operadores y uso de paréntesis en una expresión.

Hemos visto que una *expresión* es una fórmula que combina valores (constantes o variables) designados como *operandos*, y símbolos de operaciones (suma, resta, etc.) designados como *operadores*. Cuando el intérprete Python encuentra una expresión, *la evalúa* (es decir, obtiene el resultado de la fórmula) y deja disponible el resultado para ser almacenado en una variable o para ser visualizado en consola de salida o para el propósito que disponga el programador: no es obligatorio que el resultado de una expresión se asigne en una variable. Considere el siguiente ejemplo:

```
a = int(input('A: '))
b = int(input('B: '))

# 1.) expresión asignada en una variable...
c = a + 2*b
print('C:', c)

# 2.) expresión directamente visualizada (sin asignación previa)...
print('D:', 3*a - b/4)

# 3.) expresión libre... el resultado se pierde...
a + b + a*b
```

En el script anterior, el bloque 1.) muestra la forma clásica de usar y asignar una expresión: el intérprete evalúa primero el *miembro derecho* de la asignación ( $a + 2*b$ ), obtiene el resultado, y asigna el mismo en la variable *c*. En el bloque 2.) aparece otra expresión ( $3*a - b/4$ ) la cual también es evaluada por el intérprete, pero al obtener el resultado el mismo es enviado directamente a la función *print()* para que se muestre en consola de salida (una vez visualizado, el valor se pierde: ninguna variable retiene ese valor). Y en el bloque 3.) se muestra



una expresión correctamente escrita ( $a + b + a*b$ ) que será efectivamente evaluada por el intérprete sin producir error alguno, pero el valor obtenido no será visualizado y simplemente se perderá ya que no fue asignado en ninguna variable. Puede parecer que esto último no tiene sentido ni utilidad, pero por ahora lo rescatamos sólo como una posibilidad válida.

En general, si el resultado de una expresión es un número, entonces esa expresión se conoce como una *expresión aritmética*. Las tres expresiones que mostramos en el ejemplo anterior, son *aritméticas*. Note que es perfectamente posible que el resultado de una expresión sea un valor lógico (o *booleano*: *True* o *False*) en cuyo caso la expresión se conoce como *expresión lógica*; y aun más: una expresión podría dar como resultado una cadena de caracteres o algún otro tipo compuesto de resultado (en estos casos, no hay nombres específicos para ellas... son simplemente expresiones). Volveremos sobre las expresiones lógicas y las expresiones en general en Fichas posteriores.

Para resaltar la importancia del tema siguiente, mostramos ahora otros ejemplos de expresiones aritméticas. Suponga que las variables  $a$  y  $b$  son inicializadas con los valores 10 y 7 respectivamente, Tómese un par de minutos para hacer un análisis minucioso de cada una de las cuatro expresiones que siguen y trate de predecir el valor que terminará asignándose en cada una de las variables  $c$ ,  $d$ ,  $e$  y  $f$ :

```
a, b = 10, 7
c = a + 10 - 5 * b + 4
d = 3 * a + 1 - b // 4 + a % 2
e = 2 * a + b - 1 + a ** 2
f = a + b // a - b
```

La respuesta correcta es  $c = -11$ ,  $d = 30$ ,  $e = 126$  y  $f = 3$  (lo que puede fácilmente verificarse ejecutando las instrucciones una por una en el shell y mostrando sus resultados). Si no llegó a estos mismos resultados, muy posiblemente se deba a que aplicó en forma incorrecta lo que se conoce como *precedencia de ejecución* de los operadores [3] [4]. En todo lenguaje de programación, los operadores de cualquier tipo tienen diferente prioridad de ejecución en una expresión, y a esa prioridad se la llama *precedencia de ejecución*.

Cuando el intérprete Python analiza una expresión (en este caso aritmética) el operador *potencia* ( $**$ ) se aplica primero. Luego se aplican los operadores de *multiplicación* ( $*$ ), *división* ( $//$ ,  $/$ ) y *resto* ( $\%$ ). Y finalmente se aplican los operadores de *suma* ( $+$ ) y *resta* ( $-$ ). Se dice entonces que el *operador*  $**$  tiene *mayor precedencia* que los operadores  $*$ ,  $//$ ,  $/$ ,  $\%$ , y que estos últimos tienen a su vez *mayor precedencia* que los operadores  $+$  y  $-$ . Obtenidos todos los resultados de los operadores de precedencia mayor, se aplican sobre esos resultados los operadores de precedencia menor, en orden decreciente de precedencia. Por lo tanto, si tenemos  $a = 10$  y  $b = 7$  entonces la expresión

```
c = a + 10 - 5 * b + 4
```

aplicará primero la multiplicación  $5 * b$  (con resultado igual a 35) y luego hará las sumas y las restas, llegando al resultado final de -11 (obviamente, la suma  $a + 10$  se ejecuta directamente ya que no hay operadores de precedencia mayor en ella):

```
c = 20 - 35 + 4
c = -11
```

Exactamente el mismo resultado se obtendría si el producto  $5 * b$  fuese encerrado entre paréntesis (y por lo tanto, si eso es lo que quería obtener el programador, los paréntesis no son necesarios en esta expresión):



```

c = a + 10 - (5 * b) + 4
c = 20 - 35 + 4
c = -11

```

Si dos operadores tienen el mismo nivel de precedencia, se ejecutan y aplican en orden de aparición, de izquierda a derecha (y esto se conoce como *asociatividad izquierda*). En el ejemplo que sigue, asumamos las variables  $a = 2$ ,  $b = 4$ ,  $c = 5$  y  $d = 3$ :

$$e = a * b // c * d$$

Como los operadores  $*$  y  $//$  tienen el mismo nivel de precedencia y aparecen en la misma expresión, *se aplican (o asocian) de izquierda a derecha*: primero se hace  $a * b$ , con resultado 8. Ese 8 se divide en forma entera por  $c$ , con resultado 1, y ese 1 se multiplica por  $d$ , con resultado final (guardado en  $e$ ) de 3.

La excepción a esta regla es el operador  $**$  (para exponenciación). Este operador tiene *asociatividad derecha*. Si aparecen varios de estos operadores en la misma expresión, entonces se aplican *asociándolos de derecha a izquierda*. Veamos la siguiente expresión:

```

a = 5
e = a ** 2 ** 3

```

Aquí se calcula primero  $2 ** 3$  (que da 8), y luego se calcula  $a ** 8$ , con resultado  $e = 390625$ . Esto es estrictamente equivalente a haber escrito la expresión en esta forma:

$$e = a ** (2 ** 3)$$

Y note que por lo tanto, lo anterior **no es lo mismo** que lo que sigue:

```

e = (a ** 2) ** 3
# resultado: e = 15625

```

Los paréntesis siempre pueden usarse para cambiar la precedencia de ejecución de algún operador [3] y lograr resultados diferentes según se necesite. *Cualquier expresión encerrada entre paréntesis se ejecutará primero*, y al resultado obtenido se le aplicarán los operadores que queden según sus precedencias y asociatividades.

En la expresión  $c = a + 10 - (5 * b) + 4$  los paréntesis *no son necesarios* ya que en forma natural el producto  $5 * b$  se ejecutaría primero de todos modos. Pero si el programador hubiese querido multiplicar la suma  $b + 4$  por 5, entonces *debería usar paréntesis para alterar la precedencia*, en la forma siguiente:

```

c = a + 10 - 5 * (b + 4)
c = 20 - 5 * 11
c = 20 - 55
c = -35

```

Del mismo modo, si las expresiones originales:

```

d = 3 * a + 1 - b // 4 + a % 2
e = 2 * a + b - 1 + a ** 2
f = a + b // a - b

```

se reescribiesen así:

```

d = (3 * a + 1 - b) // (4 + a) % 2
e = 2 * (a + b) - (1 + a) ** 2
f = (a + b) // (a - b)

```

entonces con  $a = 10$  y  $b = 7$  los resultados obtenidos serían:

```
d = (3 * 10 + 1 - 7) // (4 + 10) % 2
d = (30 + 1 - 7) // 14 % 2
d = 24 // 14 % 2    # primero el de más a la izquierda...
d = 1 % 2
d = 1

e = 2 * (10 + 7) - (1 + 10) ** 2
e = 2 * 17 - 11 ** 2
e = 34 - 121
e = -87

f = (10 + 7) // (10 - 7)
f = 17 // 3
f = 5
```

Asegúrese de comprender cada secuencia de cálculo que hemos mostrado. Recuerde que la suma y la resta tienen precedencia menor pero que esa precedencia puede alterarse en forma arbitraria si se usan paréntesis.

En base a lo expuesto hasta aquí, debe entenderse que el conocimiento acerca de la precedencia de los operadores y el correcto uso de paréntesis resulta imprescindible para evitar cometer errores en el planteo de una expresión: en muchas ocasiones los programadores escriben una fórmula *que no querían* por aplicar mal estos conceptos y sus programas entregan resultados totalmente diferentes de los que se esperaba.

Para cerrar esta sección, mostramos la forma de escribir en Python las instrucciones que corresponden a las siguientes fórmulas (o ecuaciones) generales:

Figura 7: Uso de paréntesis para el planteo en Python de distintas expresiones aritméticas comunes.

Fórmula general	Expresión en Python	Observaciones
$p = \frac{c1 + c2 + c3}{3}$	<code>p = (c1 + c2 + c3)/3</code>	<ul style="list-style-type: none"> <li>Promedio real <b>p</b> de los valores <b>c1</b>, <b>c2</b> y <b>c3</b>.</li> </ul>
$s = \frac{n * (n + 1)}{2}$	<code>s = (n*(n+1))/2</code>	<ul style="list-style-type: none"> <li>Suma <b>s</b> de todos los números naturales del 1 al <b>n</b>.</li> <li>Puede demostrarse <b>por inducción</b> [5].</li> </ul>
$h = \sqrt{a^2 + b^2}$	<code>h = (a**2 + b**2)**0.5</code>	<ul style="list-style-type: none"> <li>Longitud de la hipotenusa <b>h</b> de un triángulo rectángulo con catetos <b>a</b> y <b>b</b>.</li> <li>Recuerde que calcular <math>\sqrt{n}</math> es lo mismo que calcular <math>n^{0.5}</math> que es <b>n**0.5</b> en Python.</li> </ul>
$p = \frac{y2 - y1}{x2 - x1}$	<code>p = (y2 - y1)/(x2 - x1)</code>	<ul style="list-style-type: none"> <li>Pendiente <b>p</b> de la recta que pasa por los puntos (<b>x1</b>, <b>y1</b>) y (<b>x2</b>, <b>y2</b>).</li> </ul>
$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$	<code>x = (-b + (b**2 - 4*a*c)**0.5) / (2*a)</code>	<ul style="list-style-type: none"> <li>Primera raíz <b>x</b> de la ecuación de segundo grado, con coeficientes <b>a</b>, <b>b</b> y <b>c</b> (con <b>a != 0</b>).</li> <li>Recuerde que calcular <math>\sqrt{n}</math> es lo mismo que calcular <math>n^{0.5}</math> que es <b>n**0.5</b> en Python.</li> </ul>





## 5.] Aplicaciones del resto de la división entera - Fundamentos de Aritmética Modular.

Hemos visto que Python provee el operador % (llamado *operador módulo* u *operador resto*) que permite calcular el resto de la división entre dos números. El estudio de las propiedades del resto forma parte de una rama de las matemáticas designada con el nombre general de *aritmética modular*, y esas propiedades tienen aplicaciones especialmente útiles en muchos campos (entre ellos, la programación y las ciencias de la computación). Expondremos brevemente algunas definiciones y notaciones propias de la aritmética modular, para pasar luego a aplicaciones de utilidad inmediata en ciertos problemas de programación.

Como primera medida, enumeraremos algunas propiedades fundamentales del resto de la división entera (el conjunto de los *números enteros* se denota aquí como  $\mathbb{Z}$ , y recordemos que incluye a los naturales, el cero, y los negativos de los naturales):

- i. El *resto de la división entera* entre  $a$  (llamado el *dividendo*) y  $b$  (llamado el *divisor*) es el primer número entero  $r$  que queda como residuo de la división parcial, tal que  $0 \leq r < b$ . Este valor indica que la división ya no puede proseguir en forma entera: el residuo parcial obtenido no alcanza a ser dividido nuevamente en forma entera por  $b$ . Todos los lenguajes de programación proveen algún tipo de operador para calcular el resto. Como vimos, en Python es el operador % [3] [4].

Ejemplos:  $14 \% 3 = 2$        $15 \% 2 = 1$        $18 \% 5 = 3$

- ii. Si se divide por  $n$  (con  $n \in \mathbb{Z}$ ) entonces pueden obtenerse exactamente  $n$  posibles restos distintos, que son los números en el intervalo  $[0, n-1]$ . El resto 0 se obtiene si la división es exacta. Pero si no es exacta, se obtendrá un resto que por definición es menor a  $n$  (pues de otro modo la división podría continuar un paso más). El valor del mayor resto posible es  $n - 1$ .

Ejemplos:

- ✓ El conjunto  $S$  de todos los restos posibles que se obtienen al dividir por  $n = 5$  contiene 5 valores:

$$S = \{0, 1, 2, 3, 4\}$$

- ✓ El conjunto  $T$  de todos los restos posibles que se obtienen al dividir por  $n = 100$  contiene 100 valores:

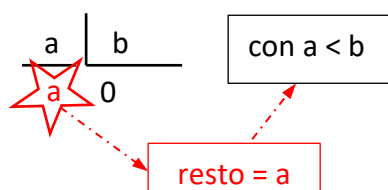
$$T = \{0, 1, 2, \dots, 98, 99\} = \{\forall r \in \mathbb{Z} \mid 0 \leq r < 100\}$$

- iii. Si el número  $a$  es divisible por  $b$  (o lo que es lo mismo, si  $a$  es múltiplo de  $b$ ) entonces el resto de dividir  $a$  por  $b$  es cero.

Ejemplos:  $14 \% 2 = 0 \Rightarrow 14$  es divisible por 2  
 $17 \% 3 = 2 \Rightarrow 17$  **no** es divisible por 3

- iv. Si el *dividendo*  $a$  es menor que el *divisor*  $b$  entonces el resto es *igual al dividendo*. Esto surge de la propia mecánica de la división entera. Si  $a < b$ , entonces el cociente  $a // b$  es 0. Para calcular el resto parcial, se multiplica ese cociente 0 por  $b$  (lo cual es 0), y el resultado se resta de  $a$ ... con lo cual es  $a - 0 = a$ . La división entera debe terminar cuando se encuentre el primer resto parcial que sea menor que  $b$ , y como  $a$  es efectivamente menor que  $b$ , aquí termina el proceso y el resto es igual al valor  $a$ , que era el dividendo.

Ejemplos:  $3 \% 5 = 3$   
 $13 \% 20 = 13$





La aritmética modular se construye a partir del concepto de *relación de congruencia* entre números enteros: Sean los números enteros  $a$ ,  $b$  y  $n$ . El número  $a$  se dice **congruente a  $b$ , módulo  $n$** , si ocurre que  $(b - a)$  es múltiplo de  $n$  (o lo que es lo mismo, si  $(b - a)$  es divisible por  $n$ ). A su vez, esto es lo mismo que decir que los números  $a$  y  $b$  tienen el mismo resto al dividirlos por  $n$ . En símbolos, la relación de congruencia se denota de la siguiente forma [5]:

$$a \equiv b \pmod{n} \quad (\text{se lee: } a \text{ es congruente a } b, \text{ módulo } n)$$

Ejemplos:

- **$4 \equiv 8 \pmod{2}$**

Se lee: 4 es congruente a 8, módulo 2. Esto es efectivamente así, ya que  $(8 - 4) = 4$ , y 4 es divisible por 2. En forma alternativa, el resto de dividir por 2 tanto al 4 como al 8 es el mismo: 0(cero).

- **$17 \equiv 12 \pmod{5}$**

Se lee: 17 es congruente a 12, módulo 5. Efectivamente,  $(12 - 17) = -5$ , y -5 es divisible por 5. Alternativamente, el resto de dividir a 17 y a 12 por 5 es el mismo: 2(dos).

En base a las relaciones de congruencia, la aritmética modular permite clasificar a los números enteros en subconjuntos designados como *clases de congruencia (módulo  $n$ )*. Dos números  $a$  y  $b$  pertenecen a la misma *clase de congruencia (módulo  $n$ )*, si se obtiene el mismo resto al dividir los números  $a$  y  $b$  por  $n$ .

Ejemplos:

- ✓ Sea  $Z$  el conjunto de los números enteros. Entonces el subconjunto  $Z_{2_0}$  de todos los números pares es una *clase de congruencia (módulo 2)*: contiene a todos los enteros que dejan un resto de 0 al dividirlos por 2 (es decir, todos los números enteros divisibles por 2):

$$Z_{2_0}: \{\dots -6, -4, -2, 0, 2, 4, 6, \dots\} = \{2 \cdot k + 0 \mid (\forall k \in Z)\}$$

- ✓ A su vez, el subconjunto  $Z_{2_1}$  de todos los números impares es otra *clase de congruencia (módulo 2)*: contiene a todos los enteros que dejan un resto de 1 al dividirlos por 2 (o sea, todos los números enteros que no son divisibles por 2):

$$Z_{2_1}: \{\dots -5, -3, -1, 1, 3, 5, 7, \dots\} = \{2 \cdot k + 1 \mid (\forall k \in Z)\}$$

- ✓ Las posibles *clases de congruencia (módulo 3)* son las tres que siguen:

$$Z_{3_0}: \{\dots -6, -3, 0, 3, 6, 9, 12, \dots\} = \{3 \cdot k + 0 \mid (\forall k \in Z)\} \quad (\text{resto} = 0 \text{ al dividir por } 3)$$

$$Z_{3_1}: \{\dots -4, -1, 1, 4, 7, 10, \dots\} = \{3 \cdot k + 1 \mid (\forall k \in Z)\} \quad (\text{resto} = 1 \text{ al dividir por } 3)$$

$$Z_{3_2}: \{\dots -2, 2, 5, 8, 11, 14, \dots\} = \{3 \cdot k + 2 \mid (\forall k \in Z)\} \quad (\text{resto} = 2 \text{ al dividir por } 3)$$

Como se dijo (*propiedad ii del resto*) los posibles restos de dividir por  $n$  son los  $n$  valores del intervalo  $[0, n-1]$ . De allí que existan  $n$  *clases de congruencia (módulo  $n$ ) distintas*, dado el valor  $n$ .

De todo lo anterior, podemos ver que los diferentes restos de dividir por  $n$  se repiten *cíclicamente*: al dividir por  $n = 3$  (por ejemplo) y comenzando desde el 0, tenemos:

$0 \% 3 = 0$	$3 \% 3 = 0$	$6 \% 3 = 0$	... (en clase de congruencia $Z_{3_0}$ )
$1 \% 3 = 1$	$4 \% 3 = 1$	$7 \% 3 = 1$	... (en clase de congruencia $Z_{3_1}$ )
$2 \% 3 = 2$	$5 \% 3 = 2$	$8 \% 3 = 2$	... (en clase de congruencia $Z_{3_2}$ )

El nombre de *aritmética modular* proviene justamente de esta característica: el proceso de tomar el resto de dividir por  $n$  produce resultados cíclicamente iguales. El valor  $n$  se denomina *módulo* del proceso, ya que por definición, un *módulo* es una *medida que se usa como norma para valorar objetos del mismo tipo* (y en este caso, los *objetos del mismo tipo* son los números que pertenecen a la misma clase de congruencia).



La aritmética modular está presente en la vida cotidiana en al menos una situación muy conocida: la forma de interpretar un reloj analógico (de agujas). El cuadrante de un reloj de agujas está dividido en 12 secciones que indican las horas principales. Pero un día tiene 24 horas (y no 12). Por lo tanto, las primeras 12 horas del día pueden leerse en forma directa, pero las siguientes 12 (esto es, desde la 13 a la 24) deben interpretarse con *relación de congruencia módulo 12*. Así, la hora 13 corresponde (o es equivalente) a la hora 1 del reloj, ya que 13 y 1 son congruentes módulo 12 (o sea que la relación es  $13 \equiv 1 \pmod{12}$ ): ambos tienen el mismo resto (que es 1) al dividir por 12).

Entonces, ahora sabemos que 13 y 1 pertenecen a  $Z12_1$  (la clase de congruencia módulo 12 con resto 1). Y cada par de horas del reloj (de agujas) separadas por 12, pertenece a una clase de congruencia módulo 12 diferente: 12 y 24 pertenecen a  $Z12_0$ ; 13 y 1 pertenecen a  $Z12_1$ ; 14 y 2 pertenecen a  $Z12_2$  y así sucesivamente. Y el resto de la división por 12 es la hora que se debe mirar en el reloj.

Por otra parte, y siempre en el ámbito del manejo de unidades de tiempo<sup>4</sup>, una aplicación práctica simple del operador resto y las relaciones de congruencia es la de convertir una cierta cantidad inicial de segundos (*is*), a su correspondiente cantidad de horas (*ch*), minutos (*cm*) y segundos restantes (*cs*). Dado que el sistema horario internacional no tiene una subdivisión de base decimal (unidades de 10, como el sistema métrico) sino sexagesimal (unidades de 60, para los minutos y los segundos), la conversión requiere algo de trabajo con operaciones de regla de 3 y aplicaciones del resto. Analicemos esquemáticamente el proceso en base a un ejemplo:

1. Supongamos que la cantidad inicial de segundos es  $is = 8421$  segundos.
2. Dado que una hora tiene 60 minutos, y cada minuto tiene 60 segundos, entonces una hora contiene  $60 * 60 = 3600$  segundos.
3. Por lo tanto, la cantidad de horas completas *ch* que hay en *is* segundos, surge del cociente entero entre *is* y 3600. En nuestro ejemplo, el cociente entero entre 8421 y 3600 es  $ch = 2$  horas.
4. Ahora necesitamos saber cuántos segundos nos quedaron de los que inicialmente teníamos. En el ejemplo, teníamos  $is = 8421$  segundos inicialmente, pero hemos tomado el equivalente a  $ch = 2$  horas, que son  $3600 * 2 = 7200$  segundos. Es fácil ver que nos quedarían  $8421 - 7200 = 1221$  segundos para continuar el cálculo.
5. Sin embargo, notemos que no es necesario multiplicar  $ch * 3600$  para luego restar ese resultado desde *is*: si se observa con atención, el proceso que se acaba de describir está basado en una *relación de congruencia (módulo 3600)*: los múltiplos de 3600 (resto 0 al dividir por 3600) equivalen a las cantidades de horas completas disponibles. Y los totales de segundos que no son múltiplos de 3600 (como 8421) dejan un resto al dividir por 3600 que es justamente igual a la cantidad de segundos en exceso...
6. Por lo tanto, se puede saber cuál es la cantidad de horas *ch* que hay en *is* tomando el cociente entero de la división entre *is* y 3600 (como vimos), y podemos saber cuántos segundos quedan para seguir operando (*ts*) simplemente tomando el resto de la misma división. En Python puede hacer esas operaciones con el operador `//` para el cociente entero y el operador `%` para el resto. En nuestro caso, quedaría  $ch = 2$ , y  $ts = 1221$ .
7. Para saber la cantidad de minutos completos *cm* que pueden formarse con *ts* segundos (aquí *cm* sería la cantidad de minutos que no llegaron a formar una hora), se procede en forma similar: un

<sup>4</sup> En la Ficha 1 hicimos una referencia del mundo del cine para la historia de Alan Turing, y ahora que estamos tratando con unidades de tiempo hacemos otra, pero del campo de la ciencia ficción: la película *In Time* del año 2011 (conocida en Hispanoamérica como el "*El Precio del Mañana*"), dirigida por Andrew Niccol e interpretada por Justin Timberlake y Amanda Seyfried, trata sobre una imaginaria sociedad futura en la que el tiempo se ha convertido en la moneda de cambio, en lugar del dinero. Las personas entregan parte del tiempo que les queda de vida para comprar un café o para comprar ropa o un viaje... ¿Cuánto valdría un segundo de tu vida en una realidad como esa? ¿Qué harías si en tu "billetera" temporal quedasen sólo 5 o 6 segundos? ¿Cómo sería tu vida si sólo tuvieras 2 o 3 minutos disponibles en cada momento del día?



minuto tiene 60 segundos, por lo que el cociente entero entre  $ts$  y 60 entrega la cantidad de minutos buscada. En nuestro caso, el cociente entero 1221 y 60 es  $cm = 20$  minutos.

8. La cantidad de segundos que queden después de este último cálculo, es la cantidad de segundos residuales  $cs$  que falta para completar el resultado: son los segundos remanentes que no alcanzaron para formar ni otra hora ni otro minuto. Y dado que este segundo proceso muestra una *relación de congruencia (módulo 60)*, entonces la cantidad remanente de segundos es el *resto de la división entre  $ts$  y 60* (en nuestro caso, el resto de la división entre 1221 y 60, que es  $cs = 21$  segundos).
9. Resultado final para nuestro ejemplo: si se tiene una cantidad inicial de 8421 segundos, eso equivale a 2 horas, 20 minutos y 21 segundos.

Dejamos para el estudiante la tarea de implementar estas ideas en un script o programa Python completo.

## 6.] Uso de Entornos Integrados de Desarrollo (IDEs) para Python: El IDE *PyCharm*.

Si bien está claro que se puede usar el shell de Python para editar, testear y ejecutar scripts y programas, el hecho es que el uso directo del shell tiene muchas limitaciones en cuanto a comodidad para el programador. El *IDLE Python GUI* que se usó a lo largo de la Ficha 01 permite escribir y ejecutar instrucciones una por una, y eventualmente también ejecutar un script de varias líneas, pero la tarea resulta incómoda (sobre todo si se está pensando en programas mucho más extensos y complejos).

En ese sentido, más temprano que tarde los programadores comienzan a usar lo que se conoce como un *Entorno Integrado de Desarrollo* (o *IDE*, tomando las siglas del inglés *Integrated Development Environment*), que no es otra cosa que un programa más sofisticado y profesional, que incluye herramientas que facilitan muchísimo el trabajo de desarrollar programas en cualquier lenguaje. Un IDE incluye un *editor de textos completo* (que en muchos casos viene provisto con *asistentes inteligentes para predicción de escritura y para ayuda contextual*), opciones de configuración de todo tipo, y herramientas de compilación, depuración y ejecución del programa que se haya escrito.

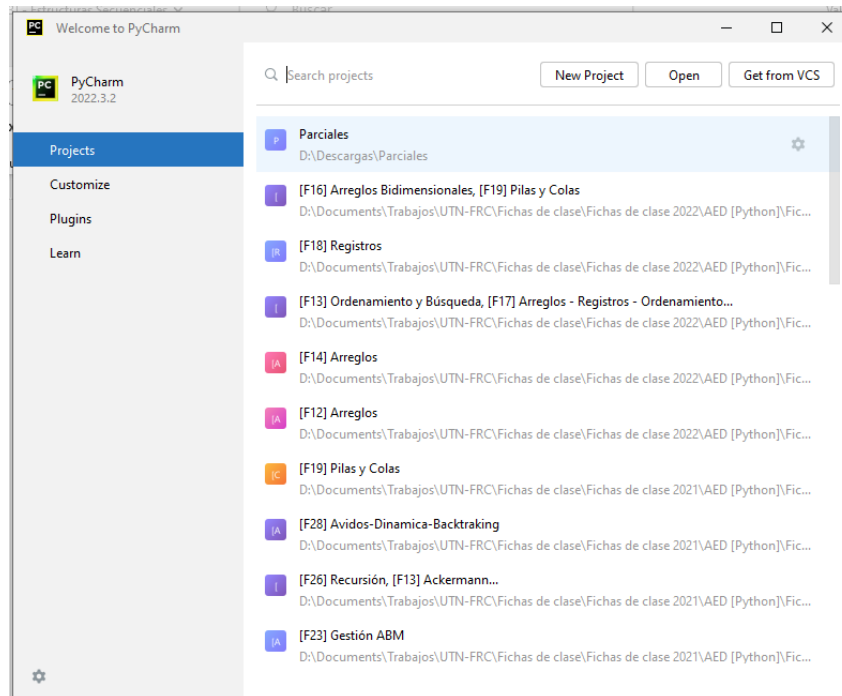
Existen numerosísimos IDEs para Python. Muchos de ellos son de uso libre y otros requieren la compra del producto y/o su licencia de uso. Algunos IDEs son simplemente editores de texto especializados para programación en múltiples lenguajes. En lo que respecta al desarrollo de la asignatura AED, y luego de un proceso de exploración y prueba que llevó cierto tiempo, nos hemos inclinado por el IDE *PyCharm* en su versión *Community Edition*, desarrollado por la empresa *JetBrains*. Las directivas de descarga e instalación han sido oportunamente provistas a los alumnos en un documento separado, dentro de los materiales subidos al aula virtual del curso en la primera semana (ver *Zona Cero* del aula virtual: *Instructivo de Instalación: Python – PyCharm*).

La versión *PyCharm Community Edition* es de uso gratuito, y aunque tiene limitaciones en cuanto a disponibilidad de recursos (con respecto a la versión *Professional*), el hecho es que dispone de muchos módulos y prestaciones que la hacen muy aplicable al contexto de un curso de introducción a la programación en Python. De aquí en más, simplemente nos referiremos a *PyCharm*, sin necesidad de aclarar que estamos usando la versión *Community*.

Para comenzar a trabajar, veamos paso a paso la forma de escribir y ejecutar un programa simple a través de *PyCharm*. Arranque el programa *PyCharm* desde el escritorio de su computadora. Si suponemos que es la primera vez que se ejecuta ese programa en esa computadora, usted verá una ventana de apertura similar a la que se muestra en la captura de pantalla de la *Figura 8*. Si no aparece esta ventana y en su lugar se ve directamente el escritorio de trabajo de *PyCharm*, se debe a que el IDE fue utilizado recientemente para desarrollar algún programa, y ese programa está cargado en ese momento (*PyCharm* automáticamente carga el último proyecto en el que se estaba trabajando antes de cerrar el IDE). Si este fue el caso, y sólo por esta vez para que pueda seguir el proceso completo desde el inicio, busque en la barra de opciones del menú de *PyCharm* la opción *File*, y dentro del menú

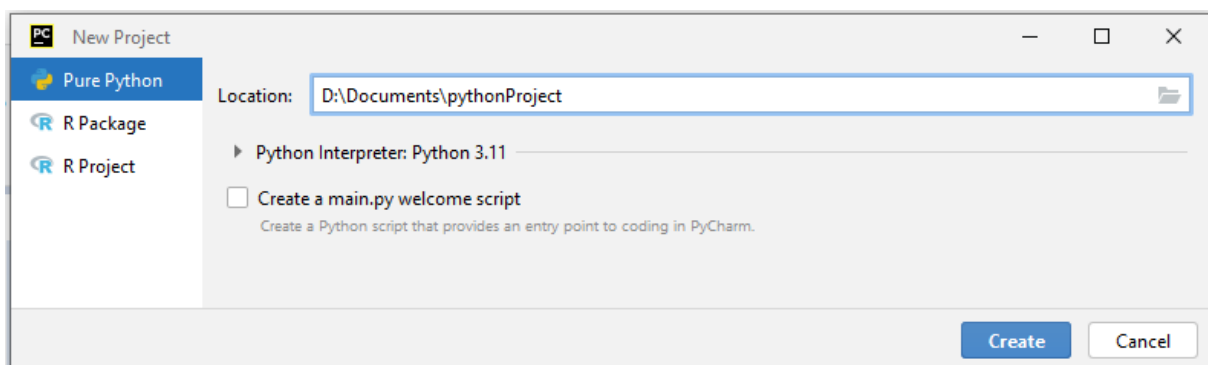
desplegable que se abre elija ahora *Close Project*. Esto cerrará el proyecto y luego de un instante verá la ventana de apertura.

**Figura 8: Ventana de apertura de PyCharm.**



En el panel central de esta ventana verá una lista de los proyectos recientemente abiertos (*Recent Projects*). En el panel de la izquierda se muestra un conjunto de opciones para configurar el entorno. Y en la parte de arriba verá tres botones: *New Project*, *Open* y *Get from VCS*. Por ahora, solo nos interesa el botón *New Project*. Al hacer click en él, la ventana cambia para mostrar los diferentes *tipos de proyectos* que puede crear (los tipos que verá dependen de qué plugins y lenguajes asociados a PyCharm tenga instalados en su computadora). Mínimamente, debería ver el tipo *Pure Python* (que es el que necesitamos). Seleccione *Pure Python* y el panel derecho mostrará un aspecto similar al de la captura de pantalla en la *Figura 9*.

**Figura 9: PyCharm - Ventana de selección de intérprete y carpeta del proyecto.**




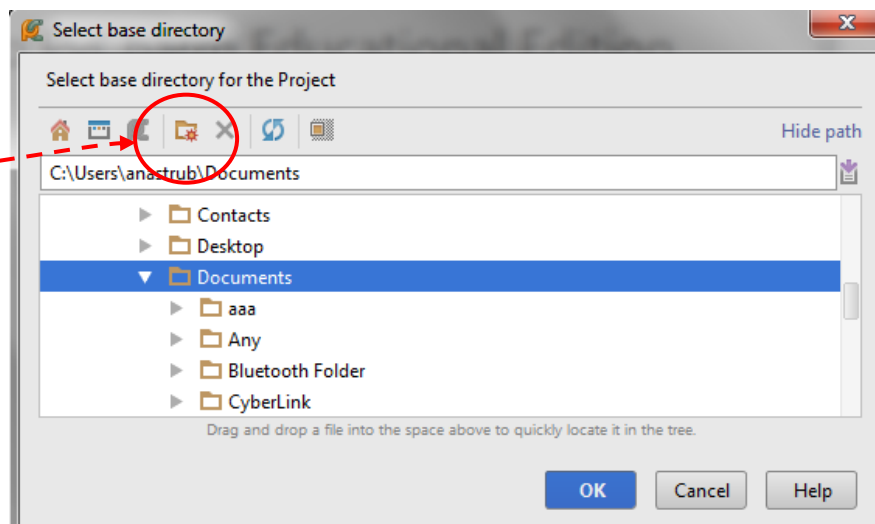
El cuadro de texto llamado *Location* le permite crear y/o elegir la *carpeta donde será alojado el proyecto* que se quiere crear. A la derecha del cuadro *Location* hay un pequeño botón con el ícono de una *carpeta abierta* () . Presione este botón y verá una ventana para navegar en el sistema de archivos de su disco local. Elija la carpeta donde usted querrá que se aloje el proyecto, y seleccione el ícono que se indica en la gráfica de la *Figura 10* para crear una carpeta específica para su proyecto.

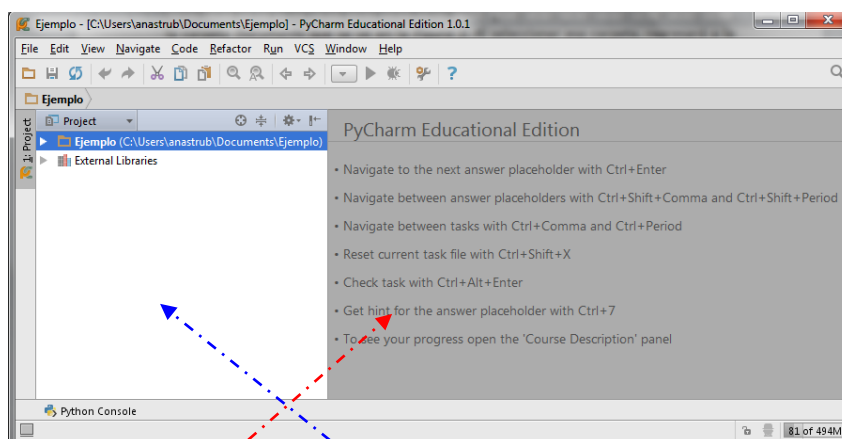
Figura 10: Creación de una carpeta específica para el proyecto.

Al pulsar este ícono, podrá crear una carpeta nueva dentro la carpeta que esté seleccionada (dentro de **Documents** en el caso del ejemplo)



Supongamos entonces que hemos creado una nueva carpeta llamada *Ejemplo* dentro de la carpeta *Documents* que se ve en la Figura 10. Al seleccionar esa carpeta, regresará a la ventana que muestra en la Figura 9, y verá que el cuadro *Location* contiene ahora la ruta de esa carpeta. Presione el botón *Create* que se encuentra debajo y a la derecha de la ventana de apertura y se abrirá el escritorio de trabajo de *PyCharm*, mostrando un aspecto semejante al de la Figura 11 (los textos que aparezcan en el bloque gris de la derecha pueden variar dependiendo de la versión de *PyCharm* que tenga instalada).

Figura 11: Escritorio de trabajo de PyCharm [con un proyecto vacío]

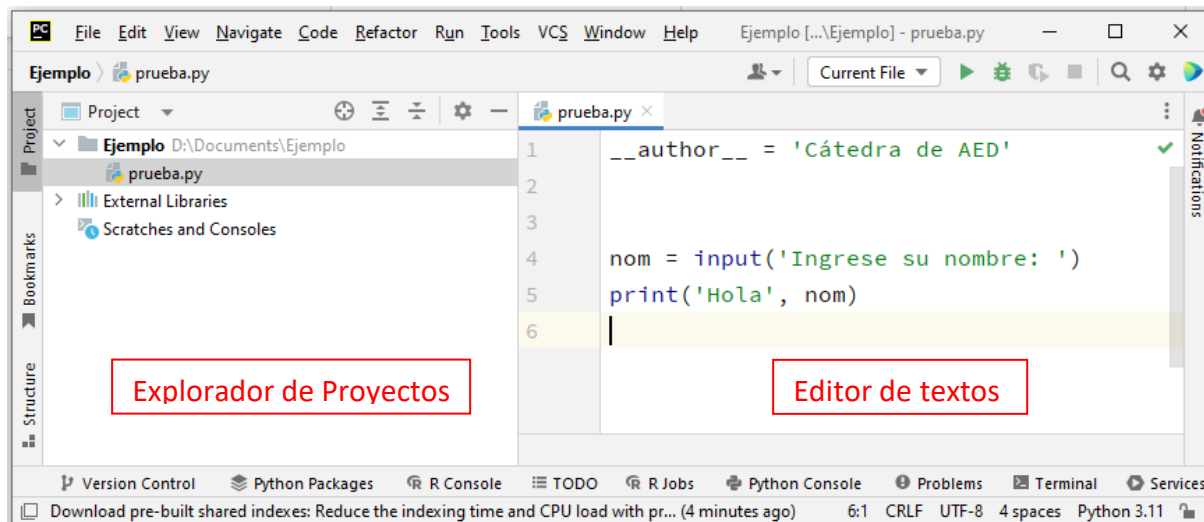


El bloque que se ve en **color gris** es la zona donde estará ubicado el editor del textos de *PyCharm* (mediante el cual podrá escribir sus programas), pero para acceder al editor debe tener al menos un programa en desarrollo dentro del proyecto. Para abrir el editor y comenzar a escribir su primer programa, apunte con el mouse a cualquier lugar del **panel blanco** (conocido como *explorador de proyectos*) que se encuentra a la izquierda de la zona gris del editor, haga *click con el botón derecho*, y en el menú contextual que aparece seleccione la opción *File*, y luego el ítem *Python File*. Se le pedirá que escriba el nombre del archivo donde será grabado el código fuente de su programa: escriba (por ejemplo) el nombre *prueba* y presione *Enter*. En ese momento, se abrirá el editor de textos (el bloque



de color gris cambiará a color blanco) quedando a la espera de que el programador comience a escribir (ver Figura 12). Note que en la captura de la Figura 12 ya estamos mostrando un pequeño programa en el bloque del editor, y suponemos que el estudiante a continuación escribirá ese programa en su propio entorno:

Figura 12: Escritorio de trabajo de PyCharm [con el editor abierto y listo para usar]



Note que *PyCharm* (dependiendo de la versión que esté usando) puede colocar una primera línea comenzando con la palabra `'__author__'` en el archivo para documentar el nombre del autor del programa. Esta es una convención de trabajo del lenguaje Python: no se preocupe por ahora de esa línea (si es que aparece). A continuación escriba un script muy simple, para cargar por teclado el nombre del usuario y mostrarle un saludo. El código fuente debería quedar así (si la línea `'__author__'` no aparece por default, simplemente agréguela para mantener el modelo completo):

```
__author__ = 'Cátedra de AED'

nom = input('Ingrese su nombre: ')
print('Hola', nom)
```

Las dos líneas en blanco que hemos dejado en el ejemplo entre la primera línea y la segunda línea están allí por razones de claridad (y en determinadas circunstancias se espera que figuren por convención de trabajo de Python). Puede dejarlas así o eliminarlas si lo desea ya que no afectará al correcto funcionamiento del script.

Ahora puede ejecutar el script: para hacerlo, haga click con el botón derecho del mouse en cualquier lugar del bloque del editor donde está escrito el programa, y cuando se abra el menú desplegable busque y presione el ícono en forma de punta de **color verde** (►) (que estará acompañado por el texto *“Run prueba”*). Observará que en la parte inferior del escritorio de *PyCharm* se abre un nuevo marco (conocido como el marco de la *consola de salida* de *PyCharm*) y allí irán apareciendo los mensajes del programa. También allí deberá el usuario escribir un nombre cuando aparezca el cursor titilante. La Figura 13 (página 50) muestra la forma en que todo se vería cuando termine de ejecutar el script, suponiendo que el nombre cargado fue *Ana*.

Un detalle muy importante en Python, es que el programador **debe** respetar el *correcto encolumnado de instrucciones* (o **indentación de instrucciones**) en su programa [3] [4], pues de otro modo el intérprete lanzará un error. La indentación o encolumnado le indican al programa qué instrucciones están en el mismo nivel o *línea de flujo de ejecución*; y en Python es **obligatorio** respetar ese



encolumnado. El script anterior funcionará sin problemas pues además de no tener errores de sintaxis, está correctamente indentado. Sin embargo, observe el siguiente ejemplo:

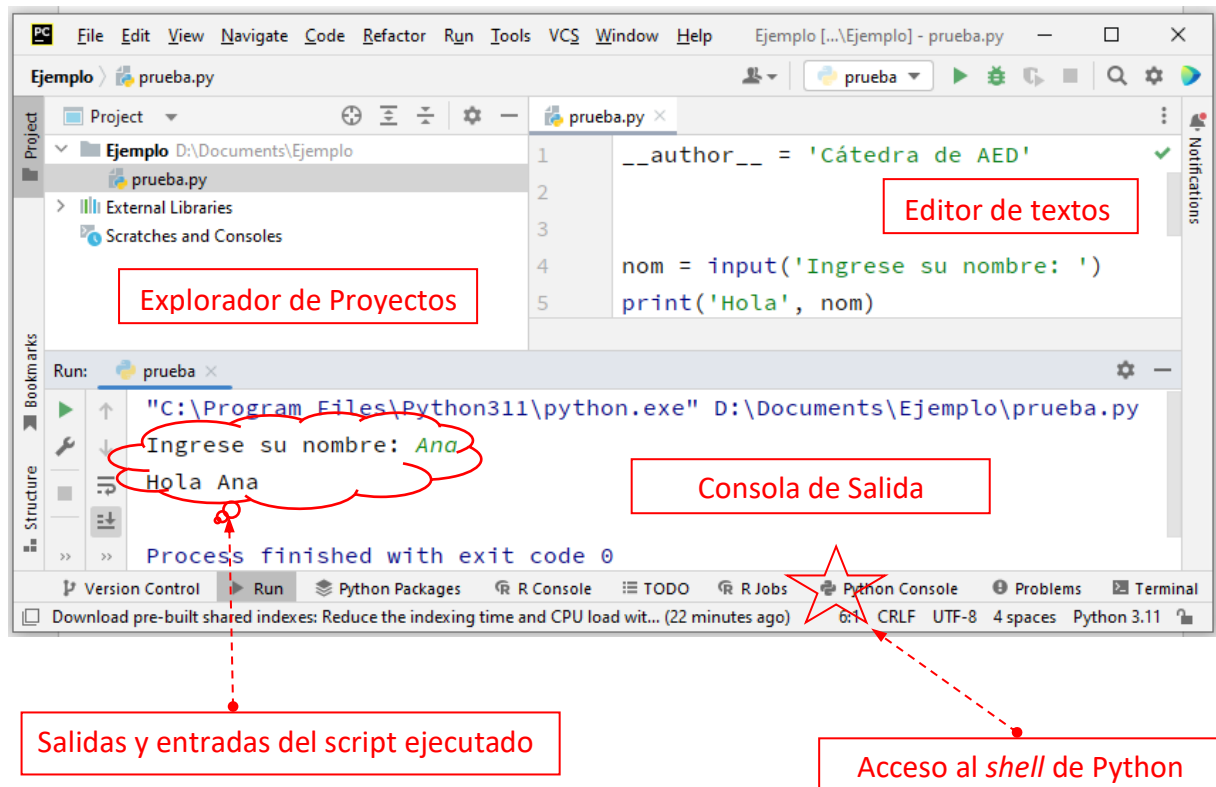
```
__author__ = 'Cátedra de AED'

nom = input('Ingrese su nombre: ')
    print('Hola', nom)
```

Se trata exactamente del mismo script original, pero ahora la tercera instrucción está fuera de encolumnado (se dice que está *mal indentada*): la instrucción `print('Hola', nom)` está corrida unos tres espacios hacia la derecha, y ya no está en la misma columna que la instrucción anterior `nom = input('Ingrese su nombre: ')`. Así como está escrito, este script no podrá ejecutarse: el intérprete Python lanzará el siguiente error en la consola de salida cuando se intente su ejecución:

IndentationError: unexpected indent

Figura 13: Escritorio de trabajo de PyCharm [luego de ejecutar un script].



De hecho, incluso antes de intentar ejecutar el script, el editor de textos de *PyCharm* detectará el error y lo marcará, subrayando con una cejilla de color rojo la letra p de la función `print()`. Además, le mostrará un mensaje aclaratorio cuando pase el puntero del mouse sobre esa instrucción. En general, el editor de textos de *PyCharm* dispone de esta característica de *predicción de errores*, de forma que el programador puede saber que está cometiendo uno antes de intentar ejecutar.

Para finalizar esta introducción al procedimiento de uso de la herramienta *PyCharm*, vuelva a la Figura 13 y observe el pequeño ícono con la leyenda *Python Console* ubicado en la parte inferior del editor de *PyCharm*. Si pulsa este ícono notará que el marco inferior (en donde hasta ahora se mostraban las entradas y salidas del script ejecutado) cambia y se muestra en su lugar el *shell de comandos de Python* (lo cual puede comprobar porque aparece el prompt `>>>` característico del ese shell). Si el shell está



abierto, obviamente, puede escribir y ejecutar órdenes directas de la misma forma en que lo hacía con el IDLE GUI que analizamos en la Ficha 01 (de hecho, es exactamente el mismo programa, pero ahora incrustado dentro del sistema de ventanas (o interfaz de usuario) de PyCharm. Puede volver a ver la consola de salida normal seleccionando ahora el ítem *Run* que figura también en la barra de la parte inferior. Dejamos para el alumno la tarea de explorar el funcionamiento general del shell "incrustado", repitiendo ejemplos vistos en la Ficha 01 o introduciendo las instrucciones que desee para que se convenza que se trata del mismo viejo shell de la Ficha 01.

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] V. Frittelli, D. Serrano, R. Teicher, F. Steffolani, M. Tartabini, J. Fenández and G. Bett, "Uso de Python como Lenguaje Inicial en Asignaturas de Programación", in *Libro de Artículos Presentados en la III Jornada de Enseñanza de la Ingeniería - JEIN 2013*, Bahía Blanca, 2013.
- [3] Python Software Foundation, "Python Documentation", 2021. [Online]. Available: <https://docs.python.org/3/>.
- [4] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [5] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.