1. Lenguajes de programación

Como lo indica la palabra, un *lenguaje de programación* es fundamentalmente *un lenguaje* de los denominados **formales**. Sus reglas son más estrictas que las del *lenguaje natural* que se utiliza habitualmente para la comunicación entre personas. Estos tienen la característica de permitir a los programadores codificar algoritmos que serán procesados de alguna manera para que una computadora pueda ejecutarlos.

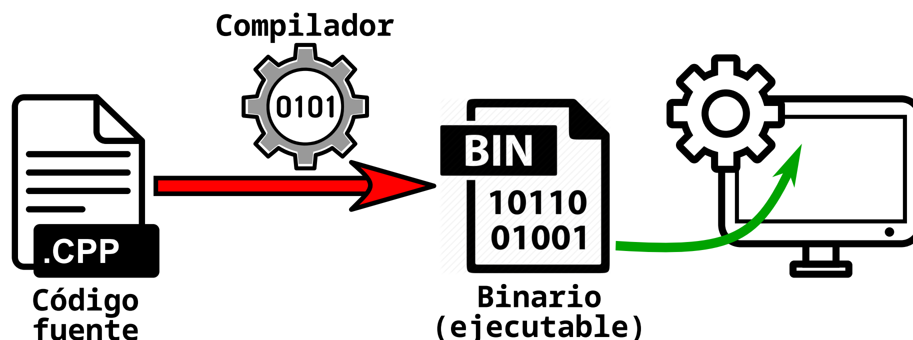
Así como cualquier lenguaje tiene sus reglas gramaticales:

- **Léxico:** conjunto de palabras o códigos (combinación de símbolos) que son válidos en el lenguaje. Por ejemplo, `casa` es un código válido en la lengua española y `c454` sería una combinación inválida. Al conjunto finito de códigos válidos se los conoce en el lenguaje de programación como **palabras reservadas** (*keywords*).
- **Sintaxis:** son las combinaciones gramaticalmente correctas para la formación de oraciones con las palabras validas y la construcción de unidades superiores. En un lenguaje de programación a estas oraciones se las conocen como **sentencias** y **expresiones**.
- **Semántica:** es el sentido o significado de las *expresiones* o *sentencias* para validarlas.

Los programas se escriben en archivos de *texto plano* que son diferentes a los que tienen formato como podrían ser los creados con un procesador de texto, donde puede cambiarse las fuentes, dar resaltado a voluntad (negritas, itálicas, subrayado, etc.). Estos archivos solo contienen texto y pueden ser escritos prácticamente con cualquier software de edición de texto. Aunque existen softwares especialmente dedicados para escribir en lenguajes de programación y tienen características que facilitan dicha tarea.

1.1. Compilado vs. interpretado

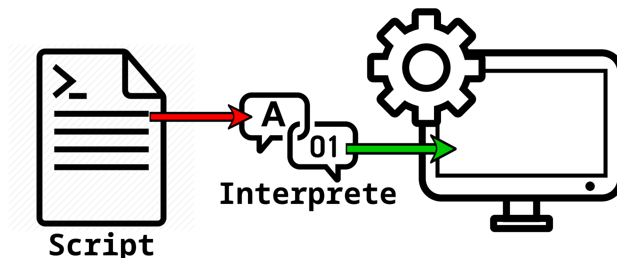
Existen, esencialmente, dos maneras en que las computadoras ejecutan los programas. La primera es utilizando **un compilador**, este es un software que traduce en su totalidad el código a **lenguaje máquina** (instrucciones en binario) que es lo que puede procesar la computadora.



Por otro lado, existen otros softwares denominados **interpretes**, que leen sentencia por sentencia y analizan expresión por expresión, decodificando a lenguaje máquina en

Matías S. Ávalos© CC BY-NC-SA 4.0

el momento (en tiempo real) cada una de las lineas de los programas. Usualmente a los lenguajes que son "interpretados" se los denomina también *lenguajes de scripting*.



Como analogía, podríamos entender que un libro puede ser traducido del inglés y ser editado en castellano. En ese caso el libro es "compilado", es posible acceder a la obra completa de una vez y el tiempo demorado está impuesto por nuestra capacidad de lectura. En cambio, si una persona toma el libro original en inglés y empieza a leerlo en voz alta en castellano, diremos que "interpreta" el libro. En este caso el tiempo demorado estará dictado por la capacidad del *interprete* en traducir y la comunicación oral, que suele ser más lenta.

Los softwares que son compilados suelen estar más optimizados en velocidad y tamaño, pero llevan más tiempo elaborarlos, porque tienen reglas más estrictas para el programador. En cambio, los softwares interpretados suelen ser menos óptimos en tiempo y uso de recursos, pero en contrapartida suelen escribirse en lenguajes de programación más ágiles y sencillos para el programador.

1.2. Clasificación en Software privativo, Open Source y Libre

Al archivo o conjunto de archivos que contienen el o los algoritmos codificados (*programa codificado*) se lo denomina **código fuente**. Es la receta de cocina para *construir* el programa ejecutable. Dependiendo si el o la autora o autores del software deciden compartir el código fuente se clasificará en: **Software Privativo** en caso de que no lo hagan o **Software de Código Abierto** (*Open Source*) si deciden hacerlo.

Para que un Software sea denominado **Software Libre**, debe cumplir con las siguientes libertades:

0. La libertad de ejecutar el programa como se desee, con cualquier propósito.
1. La libertad de estudiar cómo funciona el programa, y cambiarlo para que haga lo que se desee.
2. La libertad de redistribuir copias para ayudar a otros.
3. La libertad de distribuir copias de sus versiones modificadas a terceros. Esto le permite ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones.

Para cumplir con las libertades 1 y 3 se debe tener acceso al código fuente, es decir, que el *Software Libre es de código abierto*, pero no necesariamente esta relación se da al revés.

Matías S. Ávalos© CC BY NC SA 4.0

1.3. Paradigmas

Los paradigmas de programación son maneras de pensar o *"encarar"* los algoritmos. Estos difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema.

Se presenta a continuación una clasificación *clásica* de los paradigmas, teniendo en cuenta que existen diferencias entre bibliografías.

1. **Paradigma Imperativo:** se basa en dar instrucciones a la computadora de *cómo hacer* las cosas en forma de algoritmos, en lugar de describir el problema o la solución. Dentro de éste podemos diferenciar:
   a) **Procedural u orientado a procesos:** este paradigma permite listar de forma estructuradas las operaciones necesarias a ser programadas, es posible agrupar estas ordenes en procedimientos y funciones, así como estructurar datos complejos o interconectados.
   b) **Orientado a objetos:** aquí se intenta reflejar la realidad, se describen clases que modelizan objetos y estos interactúan transmitiendo mensajes entre sí para solucionar el problema modelizado.
   c) **Orientado a eventos:** en vez de seguir un flujo secuencial de instrucciones, se espera a que *eventos* disparen porciones de código. Esto pude aplicarse tanto en programación procedural como en orientada a objetos.
2. **Paradigma declarativo:** está basado en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones.
   a) **Funcional:** está basada en la definición los predicados y es de corte más matemático.
   b) **Lógico:** se basa en el concepto de función, gira en torno al concepto de predicado, o relación entre elementos.
3. **Multiparadigmas:** hoy en día gran parte de los lenguajes de programación soportan diferentes paradigmas. Tienen un paradigma base o raíz, pero fracciones de código pueden estar escritos en otros por cuestiones de síntesis, facilidad al momento de planteare esa parte del algoritmo, etc.

1.4. Niveles

Oficialmente existen 2 clasificaciones de nivel para los lenguajes de programación: **bajo** y **alto**. Esto se refiere a la relación entre el lenguaje y la máquina (computadora). Los lenguajes de bajo nivel son los que están íntimamente relacionados con las instrucciones que puede ejecutar una computadora en particular y son el *lenguaje máquina* (binario) y el *assembler*. Cada procesador tiene su propio set de instrucciones y su propio assembler, por lo tanto, el programador debe tener un conocimiento muy detallado sobre el procesador que está programando y el sistema operativo donde va a ejecutarse su programa.

Los lenguajes de *alto nivel* abstraen al programador de la máquina, y pueden utilizarlos sin saber siquiera sobre que computadora correrá el programa. Ya que de eso se encarga el

Matías S. Ávalos© CC BY NC SA 4.0

compilador, que traduce todo el código a un lenguaje máquina específico, o del interprete que debe estar preparado para esa computadora en particular y traducir las instrucciones del script.

Hay un neologismo que define lenguajes de *nivel intermedio*. Estos son los lenguajes de alto nivel que tiene facilidades para acceder a instrucciones de bajo nivel o que su abstracción no es tan elevada. Esto último indicaría que debemos tener consideraciones si se programa para una arquitectura especifica de computadora o sistema operativo en particular. Lo cual no ocurre con los de *alto nivel* en esta clasificación, los cuales pueden compilarse o ejecutarse en cualquier sistema sin la necesidad de tener consideraciones particulares.

2. Programación en `Python`

Este lenguaje de programación o scripting inicialmente interpretado y creado por Guido van Rossum a finales de los 80's y principio de los 90's, tiene la finalidad de tener una sintaxis sencilla, pero sin dejar de ser un lenguaje potente. Su nombre está inspirado en un grupo de humor británico, los *Monty Python*[1], y es un derivado del lenguaje `ABC`, pensado para la enseñanza.

`Python` es un lenguaje de programación orientado a objetos (`POO`), pero con muchas funciones básicas que nos permitirán estudiar la programación estructurada y procedural dentro del paradigma imperativo. Cabe destacar, que como la mayoría de los demás lenguajes, hoy en día se considera multiparadigma, pero aprender a diseñar algoritmos es el principal objetivo del curso.

2.1. Salida Estándar

La mayoría de los lenguajes de programación están orientados al desarrollo de sistemas que eventualmente podrían o no tener una interfaz de usuario. Las interfaces de usuario (**UI - *User Interface***) podrían ser gráficas (**GUI**) tanto sean aplicaciones de celular, desktop, web, etc. o bien de **consola**. Ésta última es la que la mayoría de los lenguajes utiliza como salida y entrada estándar o predefinida, ya que los componentes gráficos son elementos que suelen ser más independientes o diversos (dependen del sistema operativo, biblioteca gráfica u otro elemento ajeno al lenguaje). Esta interfaz es aún ampliamente utilizada, especialmente en servidores y terminales de servicios.

2.1.1. La consola/terminal

Lo cierto es que la salida estándar suele darse por medio de un monitor o pantalla. Los primeros programas serán desarrollados para una consola o terminal de texto, también denominada interfaz de linea de comandos (**CLI - *Command Line Interface***).

[1] https://es.wikipedia.org/wiki/Monty_Python

Matías S. Ávalos© CC BY NC SA 4.0

Figura 1: Terminal de Linux

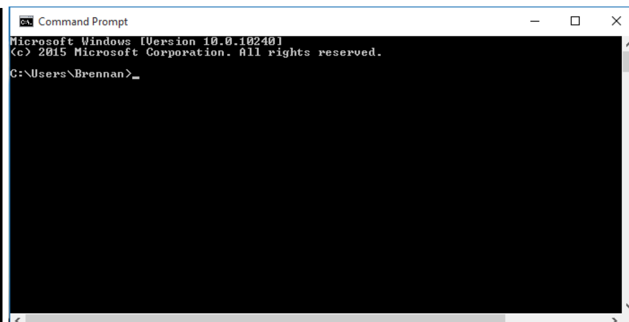


Figura 2: cmd.exe de MS Windows

2.1.2. El comando `print()` (*imprimir*)

Como se hizo mención anteriormente, `Python` nació como un lenguaje interpretado, es decir que existe un programa que lee, evalúa y muestra los resultado de los comandos continuamente. Usualmente a estos softwares se los denomina **REPL** (***Read-Eval-Pring-Loop***), y la mayoría de los lenguajes de estas características poseen uno, donde simplemente podemos escribir instrucciones y serán ejecutadas. Otra alternativa es guardar las instrucciones en un archivo de texto plano denominado *script*, y luego pedirle al interprete que ejecute todo el *lote* de instrucciones de una sola vez.

El primer comando de `Python` que se expone a continuación es el que muestra una linea de texto por la pantalla `print` el cual recibe entre paréntesis un texto encerrado entre comillas dobles o simples indistintamente. Para poder ejecutarlo debemos utilizar el interprete desde la consola o guardar el comando en un *script* que termine en `.py` para luego pedirle al interprete que lo ejecute:

Programa 1: Saludo en el `REPL`

```
1 /home/user:~$ python
2 >>> print("¡Hola!")
3   ¡Hola!
4 >>> quit()
5 /home/user:~$
```

En el Programa 1 podemos observar que en la *línea 1* se invoca al `REPL`. Esto nos habilita un entorno identificado por los tres corchetes angulares de cierre >>> en la *línea 2* donde podremos escribir la instrucción `print()` con el texto `"¡Hola!"`. Luego, como respuesta o resultado de la ejecución, veremos el texto propiamente dicho en la *línea 3* y el `REPL` nos devolverá el control para la siguiente instrucción. Cuando deseemos salir del interprete debemos hacerlo invocando a la instrucción `quit()` y con ello volveremos a la consola del sistema operativo como se observa en las *lineas 4 y 5*.

Normalmente lo que se hará es escribir las instrucciones en un archivo de texto plano (*script*) y pedirle al interprete que lo ejecute completo, aunque se utilizara oportunamente el `REPL` para probar instrucciones antes de incluirlas al script definitivo para asegurar el correcto funcionamiento del algoritmo.

Programa 2: Archivo `saludo.py`

```
1 # Muestra el texto '¡Hola!' por la pantalla
2 print("¡Hola!")
```


Matías S. Ávalos© CC BY NC SA 4.0

```
/home/user:~$ python saludo.py
¡Hola!
/home/user:~$
```

Se observa al principio del *script* `saludo.py` que las líneas que empiezan con el carácter '`#`' son lo que se conocen como *comentarios*. No son instrucciones ejecutables, sino que son aclaraciones para los programadores o personas que leerán el código fuente de nuestro programa, para facilitar la comprensión. En este caso es un *comentario de línea* ya que todo lo que sea escrito en ella será un comentario y a partir de la siguiente ya no es más parte del comentario. Existe además una manera de hacer comentarios de más de una linea, los cuales empiezan con trile comillas (dobles o simples) y terminan de la misma forma. A estos se los conoce como *comentario multilínea*.

```
"""
  Todo lo encerrado entre triple comilla
  es un comentario...
"""
```

2.2. Entrada Estándar y Variables

2.2.1. Memoria

Todas las computadoras poseen al menos una *memoria* en la que pueden almacenar **datos** y/o **instrucciones**. Las mismas son ***finitas***, por lo tanto los programas y la cantidad de datos que podemos almacenar en ellas también lo son. La memoria (por lo general), es el recurso más escaso en una computadora, por ello sería conveniente intentar hacer algoritmos pequeños y utilizar la menor cantidad de *variables* posibles. Sin embargo, no es algo de lo que nos preocuparemos por el momento.

Las **variables son espacios de la memoria** en las cuales podemos almacenar *datos* y son denominadas así, porque esos datos pueden *cambiar*. Es decir que podemos ***asignarles*** un dato y luego alterarlo según sea necesario. Estos datos pueden ser, un número entero, números con coma, un carácter (una letra) o una cadena de caracteres (*string*).

Nota

Las cadenas de caracteres son *textos* usualmente encerrados entre comillas (dobles o simples) dependiendo del lenguaje de programación. En caso de `Python` es indistinto, pudiendo utilizar las que el programador prefiera y no existen los caracteres, sino los *strings* de 1 sola letra.

Como se observa en el Programa 3, hay una variable llamada ***"nombre"*** a la cual el *usuario*, que ejecuta nuestro algoritmo debe asignarle un valor utilizando la instrucción `input()` la cual muestra un mensaje por pantalla. En ese momento nuestro programa quedará en estado de *espera* a que el usuario ingrese un valor y luego presione la tecla `ENTER` para indicar que ha finalizado el ingreso. Luego es mostrada con la instrucción `print()` combinando un *string* fijo ("`¡Hola, `") con el valor ingresado y contenido en

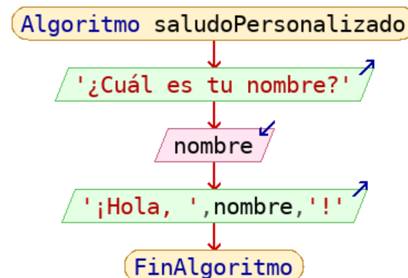
Matías S. Ávalos© CC BY-NC-SA 4.0

Programa 3: Saludo con una variable

```
1 nombre = input("¿Cual es tu nombre? ")
2 print(f"¡Hola, {nombre}!")
```

Salida del programa

```
¿Cual es tu nombre? Matías
¡Hola, Matías!
```



nombre indicado al encerrarlo entre llaves y luego finaliza con un "!". Para que el valor de *nombre* sea reemplazado en la cadena, es necesario anteponer la letra **'f'** (*format*) al string para que funcione este mecanismo.

Existen diversos **tipos de datos**, `Python` es un lenguaje de tipado dinámico, e infiere de que tipo se trata al momento de asignarle un valor con el operador '='. `input()` devuelve siempre un *string*, para poder almacenar otro tipo de dato debemos forzar la conversión de tipo utilizando las siguientes funciones:

Tipos de datos

- `int(input("Número entero: "))`
- `float(input("Número con coma: "))`
- `bool(input("Valor booleano: "))` (*true* o *false*)
- `str(2.2)` (para transformar a *string* otro tipo)

2.2.2. Dar valores a una variable

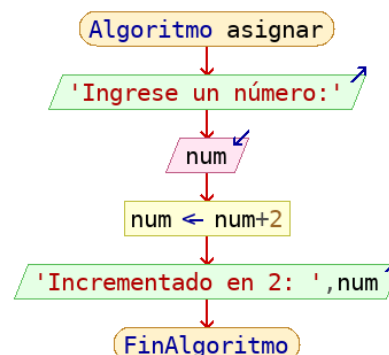
Muchas veces necesitamos modificar o "crear" datos sin depender exclusivamente del *usuario*. En ese caso, la asignación no será a través de la instrucción "**input**" sino a través del operador de asignación que en pseudocódigo suele indicarse con una flecha hacia la izquierda "←" o "<-". En `Python` se utiliza simplemente el operador "=" ubicando la variable a izquierda y lo que se asignará (lo que se guardará) en ella a la derecha.

Programa 4: Asignación

```
1 num = int(input("Ingrese un número:"))
2 num = num + 2
3 print(f"Incremento en 2: {num}")
```

Salida del programa

```
Ingrese un número: 4
Incremento en 2: 6
```



En este caso, tomamos el valor en la variable *"num"* y lo incrementamos en 2, guardando el nuevo valor en la misma variable (en el mismo espacio de memoria). Veamos el

Matías S. Ávalos© CC BY NC SA 4.0

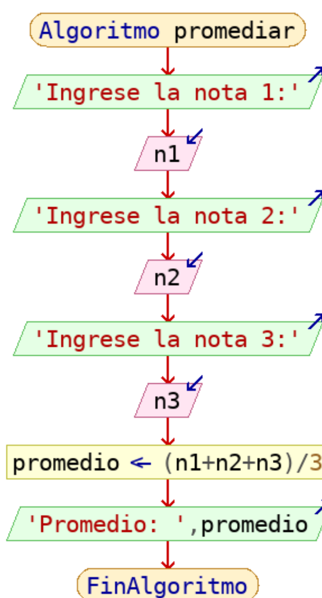
ejemplo del calculo de un promedio de notas.

Programa 5: Asignación

```python
n1 = int(input("Ingrese nota 1: "))
n2 = int(input("Ingrese nota 2: "))
n3 = int(input("Ingrese nota 3: "))
promedio = (n1+n2+n3) / 3
print(f"Promedio: {promedio}")
```

Salida del programa

```
Ingrese nota 1: 5
Ingrese nota 2: 5
Ingrese nota 3: 6
Promedio: 5.33333
```



En el Programa 5, creamos 3 variables *"n*"* al principio del programa, pero al declarar la variable *promedio* además usamos el *operador de asignación*. Es decir que **inicializamos** la variable al momento de declararla.

Nota

En el `Python` disponemos de las operaciones matemáticas más comunes:

- `+` (suma)
- `-` (resta)
- `*` (multiplicación)
- `/` (división)
- `%` (resto de la división entera)
- `**` (potencia)
- `//` (división entera)

2.3. Interpretación de enunciados

Como aclaramos en el primer encuentro, la programación es la codificación de los algoritmos en un lenguaje de programación. Pero es preciso comprender los enunciados para pensar una solución y como articular la misma, es decir *cómo* diseñaremos nuestro algoritmo (*modelo de solución*). Interpretar los enunciados es casi tan importante (*¡o más!*) que la solución a los mismos. Ya que podemos buscar a alguien que lo solucione por nosotros o que nos oriente, buscar en internet, etc.. Pero, si no comprendemos el problema o lo entendimos mal, llegaremos a soluciones erróneas o a ninguna en absoluto.

Ejemplos:

- Elabore un algoritmo que pida ingresar un número, determine su anterior y muéstrelo en pantalla.
- Elabore un algoritmo que pida ingresar un número, determine su doble y muéstrelo en pantalla.

Matías S. Ávalos© 4.0

- Elabore un algoritmo que pida ingresar un número, determine el doble del siguiente y lo muéstrelo en pantalla.
- Elabore un algoritmo que pida la cantidad de pesos argentinos para hacer una compra de divisas en dólares y muestre la cantidad que se pueden comprar.

2.4. ¿Enteros o decimales?

Al momento de codificar un algoritmo es importante definir qué tipo de variable es la que mejor se adecúa al dato que queremos almacenar. Así pues, si necesitamos almacenar un texto usaremos `string` y especialmente al momento de utilizar números debemos optar por alguna de las dos posibilidades `int` o `float`.

El hecho de elegir un tipo de dato con, o sin coma, estará dado por la naturaleza del problema a resolver por el algoritmo. Por ejemplo, si el problema es repartir una cantidad dada de caramelos entre una *x cantidad* de personas, estamos ante un claro ejemplo de uso de números enteros. No es posible, en principio, repartir $4\frac{3}{4}$ caramelos o tener $2\frac{1}{2}$ personas. Y, en caso de necesitar almacenar o procesar un dato como una temperatura en grados centígrados o el volúmen de un líquido en litros, posiblemente debamos utilizar números con decimales.

2.4.1. Operaciones entre enteros y números con decimales

Como se menciona en el cuadro al final de la sección 2.2.2, existen 3 operadores relacionados con la división entre números. Analice la salida del siguiente *script* de `Python`:

Programa 6: Operación entre enteros

```python
# Operación con enteros (int)
promedio = (5+5+6) / 3
print(f"El promedio entre 5, 5 y 6 es {promedio}")
```

```
El promedio entre 5, 5 y 6 es 5.333333333333333
```

Como se observa al dividir el resultado es un `float`, a pesar de haber operado entre enteros (`int`). Esto es lo que uno naturalmente espera de esta cuenta. Pero, ¿qué ocurre cuando son necesarios resultados enteros? Es decir, si fuera necesario repartir una cantidad de unidades, por ejemplo, caramelos entre personas, no debemos mostrar resultados como $5.\widehat{3}$ caramelos para cada uno. Allí se debe optar por una división entera:

Programa 7: Divisiones enteras y resto de la división

```python
caramelos = int(input("Ingrese cantidad de caramelos: "))
personas = int(input("Ingresa cantidad de personas: "))

car_x_persona = caramelos // personas # DIVISIÓN ENTERA
print(f"Caramelos para cada persona {car_x_persona}.")
car_repartidos = car_x_persona*personas
print(f"Caramelos repartidos {car_repartidos}.")
car_restantes = caramelos % personas  # RESTO DE LA DIVISIÓN
print(f"Caramelos no repartidos {car_restantes}.")
```


Matías S. Ávalos© CC BY-NC-SA 4.0

Una posible salida de este programa

```
Ingrese cantidad de caramelos: 50
Ingresa cantidad de personas: 3
Caramelos para cada persona 16.
Caramelos repartidos 48.
Caramelos no repartidos 2.
```

2.4.2. Redondeo y Truncamiento

Hay dos operaciones que son muy comunes dentro de las computadoras al operar y debemos utilizarlas correctamente. El redondeo y el truncamiento.

Hay cálculos donde podemos aproximar el valor desestimando los números seguidos desde la cifra decimal deseada, a esta operación se la denomina truncamiento. Simplemente consta en deshacerse de los números decimales que no son necesarios. Por ejemplo, si queremos 2 decimales para truncar, debemos multiplicar por 100 para pasar los decimales deseados a la parte entera y asignarlos a una variable de este tipo, deshaciéndonos de los decimales no deseados. Luego se vuelve a pasar a `float` dividido por 100 para devolver los decimales al lugar correspondiente.

$$23{,}12357 \times 100 \rightarrow 2312\cancel{,357} \div 100 \rightarrow 23{,}12 \tag{1}$$
$$54{,}25786 \times 100 \rightarrow 5425\cancel{,786} \div 100 \rightarrow 54{,}25 \tag{2}$$

Programa 8: Truncar a 2 decimales

```python
pi = 3.14159
print(f"PI sin truncar: {pi}")
pi = int(pi*100) / 100
print(f"PI truncado a 2 decimales: {pi}")
```

Por último, el redondeo es la técnica de aproximación numérica donde se toman en cuenta los decimales anteriores al número de decimales deseados. Por ejemplo, si deseamos 2 decimales para mostrar un resultado, entonces el proceso sería el de multiplicarlo por 100 para que esos decimales queden en la parte entera, sumar 0,5 para redondear al entero más próximo y truncar la parte decimal. Finalmente volver a dividir por 100 para retornar al número original ya redondeado:

$$23{,}12357 \times 100 \rightarrow 2312{,}357 + 0{,}5 \rightarrow 2312\cancel{,857} \div 100 \rightarrow 23{,}12 \tag{1}$$
$$54{,}25786 \times 100 \rightarrow 5425{,}786 + 0{,}5 \rightarrow 5426\cancel{,286} \div 100 \rightarrow 54{,}26 \tag{2}$$

Programa 9: Redondeo a 4 decimales

```python
pi = 3.14159
print(f"PI sin redondear: {pi}")
pi = int((pi * 10000) + 0.5) / 10000
print(f"PI redondeado a 4 decimales: {pi}")
```


Matías S. Ávalos© CC BY NC SA 4.0

2.5. Nombrando variables

Es importante utilizar nombres para las variables que indiquen claramente cual es su rol dentro del programa. Además existen diferentes convenciones. En general, acordaremos que las variables siempre deben empezar con una letra alfabética en minúscula.

Los nombres de las variables no pueden tener espacios, solo caracteres alfanuméricos (letras y números) sin utilizar vocales con tildes o `ñ`.

Si el nombre de una variable se compone con dos o más palabras puede optar por separarlas con guiones bajos[2], por ejemplo `mejor_cancion` o empezando la siguiente palabra con una letra mayúscula[3], por ejemplo `mejorCancion`.

Las variables del tipo *booleanas* deben empezar preferentemente con el verbo ser/estar, tener o poder: `es_mayor`, `estaApagado`, `esta_sano`, `pudeLeer`, `puede_escribir`, `tieneComentarios`, `tiene_subtitulos`, `esFragil`, etc. También podría ser aceptable si es obvio el contexto, pero siempre que sea posible optaremos por esta convención adoptada del inglés.

```
cantidadc # MAL:¿Qué significa la c después de 'cantidad'?
cantidad_clientes # BIEN: Es claro de qué es la cantidad
i # DEPENDE: BIEN si el uso es trivial, sino MAL
indice # DEPENDE: BIEN si es obvio de qué es el índice
puntos_totales # DEPENDE: BIEN si es obvio que se está puntuando,
    MAL si es ambiguo
_cuenta # MAL: Evite nombres que comiencen con guines bajos
cuenta # MAL: ¿Qué se está contando o es el resultado de una
    cuenta?
datos # MAL: ¿Qué tipo de datos datos?
tiempo # MAL: ¿Tiempo de qué?¿En horas, minutos o segundos?
minutosTranscurridos # BIEN: Es descriptivo dentro del contexto
valor1, valor2 # MAL: ¿valores de qué? Es dificil diferenciarlos.
cant_manzanas # BIEN: es descriptivo
monstruos_eliminados # BIEN: Descriptivo.
x, y # DEPENDE: Bien si el uso es trivial, sino MAL.
ganado # MAL: debería empezar con es, esta o puede
estaPartidaTerminada # BIEN: es claro que es una variable
    booleana y cuando es true y cuando false.
tiene_cartucho # BIEN: es claro dentro del contexto
```

3. Ejercitación

3.1. Intente resolver los siguientes problemas planteando los algoritmos en diagramas y codifíquelos en Python:

1. Ingrese un número y muestre la mitad del siguiente.

[2] Convención conocida como *snake_case*.
[3] Convención conocida como *lowerCamelCase*.

Matías S. Ávalos© CC BY NC SA 4.0

2. Ingrese 2 números y muestre la suma en pantalla.
3. Ingrese 3 números y muestre el producto de los dos primeros sumado al tercero.
4. Ingrese 2 números, muestre la diferencia entre ambos.
5. Ingrese su año de nacimiento y muestre la edad que alcanzará en 2030.
6. Ingrese la temperatura en grados centígrados y muéstrelos en grados Fahrenheit.
7. Ingrese dos ángulos internos de un triangulo y muestre el tercero.
8. Pida ingresar un cantidad de canicas **rojas**, y sabiendo que hay canicas rojas, amarillas y trasparentes, que las rojas son el doble que las transparentes y las amarillas el triple de las rojas, muestre cuantas canicas de cada color hay y el total.
9. Ingrese un número con coma, multiplíquelo por 3.125 y muestre el resultado redondeado en 2 decimales.
10. Ingrese el valor de lista de un producto y muestre el valor sumando el IVA (21 %), redondee a 2 decimales.
11. Ingresando la cotización del dólar estadounidense y la cantidad que se desea comprar, calcule y muestre la cantidad de pesos argentinos necesarios para dicha compra redondeado a 2 decimales.
12. Juan salio a comer una pizza con sus dos amigos de la primaria y desea determinar de cuánto dinero tiene que disponer cada uno. Si se reparten los gastos por igual: ingrese el total de la cuenta y determine cuánto dinero pagará cada uno.
13. Mariano quiere un algoritmo que al ingresar el dinero que posee determine cuántos alfajores puede comprar para él y sus 3 compañeros de trabajo, sabiendo que cada alfajor cuesta $15.25.
14. Pablo quiere diseñar un algoritmo en el que pueda ingresar el dinero en su bolsillo, ya que desea comprar caramelos para 3 amigos y quedarse con al menos 1 para él. Sabiendo que los caramelos cuestan $1.25, el algoritmo debe producir las siguientes salidas:
    a) Cantidad de caramelos que puede comprar.
    b) Cantidad de caramelos repartidos en total.
    c) Cantidad de caramelos que se queda cada uno de sus amigos.
    d) Cantidad de caramelos que se queda Pablo.

3.2. Indique si los nombres elegidos para las siguientes variables son correctos o no y porqué:

- `suma # Asuma que es obvio lo que se está sumando.`
- `_manzanas`

Matías S. Ávalos© CC BY NC SA 4.0

- `VALOR`
- `dinero depositado`
- `clientes_totales`
- `cantFrutas`
- `metros_de_caño`
- `metrosDeTuberia`
- `salir`
- `puedeCorrerConZapatillas`
- `puedeCorrer_descalzo`
- `esta_sobre_hielo`

Matías S. Ávalos© CC BY NC SA 4.0