Clase 1. ¿Qué es la Computación?

MIT 6.0001. Introduction to Computer Science and Programming in Python.

Resumen

Esta primera clase es una introducción tanto a lo que es la computación, su historia y cómo funciona un programa en sus partes internas, así como al lenguaje de programación Python.

1. Principales tareas de un computador.

Un computador realiza dos principales tareas: La primera es **calcular** (o computar). Algunos cálculos vienen incorporados en el lenguaje computacional, tales como las operaciones aritméticas; pero también es posible **programarlas** por nosotros mismos/as, definiéndolas de manera tal que el computador pueda realizarlas. Algo a tener en cuenta, es que **solo hará lo que le digamos y de forma literal**.

La otra tarea que realiza un computador, es **almacenar información**. Por ejemplo, los cálculos que va realizando, después los va guardando en su memoria.

2. Pensamiento Computacional.

Para saber cómo darle una instrucción a un computador, antes debemos entender el tipo de conocimiento que sigue.

Los computadores operan a partir de lo que en epistemología se conoce como **Conocimiento Imperativo** (o Procedimental), que es aquel que se expresa por medio de reglas o **instrucciones**.

Por ejemplo, si queremos calcular la raíz cuadrada de un número x en un computador, no

lo hará señalándole que "es un número y tal que $y \cdot y = x$"[1]. Más bien, tenemos que darle instrucciones para que lo haga. Usemos las definidas (aparentemente) por el matemático griego Herón de Alejandría:

1. Comience con un supuesto s_0.
2. Si $s_0 \cdot s_0$ es lo suficientemente cercano a x, pare y señale que s_0 es la respuesta.
3. En caso contrario, cree un nuevo supuesto s_1 promediando a s_0 con x/s_0. Es decir:

$$s_1 = \frac{s_0 + (x/s_0)}{2}$$

4. Repita el paso 2, pero usando a s_1 como su nuevo supuesto.

Al seguir estos pasos, veremos que el resultado irá aproximándose al valor real de la raíz cuadrada, aunque nunca será igual a éste.

El conjunto de instrucciones que acabamos de ver, consistió de tres partes:

- Una secuencia de pasos simples.
- Un flujo de control → Cada paso se ejecuta siguiendo un orden.
- Un medio para determinar cuándo debe parar (o "converger") el proceso.

Una secuencia finita de instrucciones como la vista para calcular una raíz cuadrada, recibe el nombre de **Algoritmo**. Se caracteriza por describir un cálculo (*computation*) que cuando se ejecuta sobre un conjunto de entradas (*inputs*), pasa por un conjunto de estados bien definidos produciendo un resultado (*output*).

En otras palabras, un **algoritmo** es una **descripción no ambigüa de cómo realizar un cálculo**. Podemos entenderlo como una receta, pero con pruebas para decidir cuándo un paso está completo o para saltar de una instrucción a otra, entre otras particularidades.

3. Instrucciones como Procesos Mecánicos.

Históricamente, las secuencias de instrucciones comenzaron a materializarse en **Computadores de Programa Fijo** (*fixed-program computers*). Estas máquinas eran diseñadas para **realizar una tarea específica** y solo era posible modificarlas cambiando todo su circuito por uno que hiciera la funcionalidad de interés.

Posteriormente, se crearon los **Computadores de Programa Almacenado** (*stored-program*

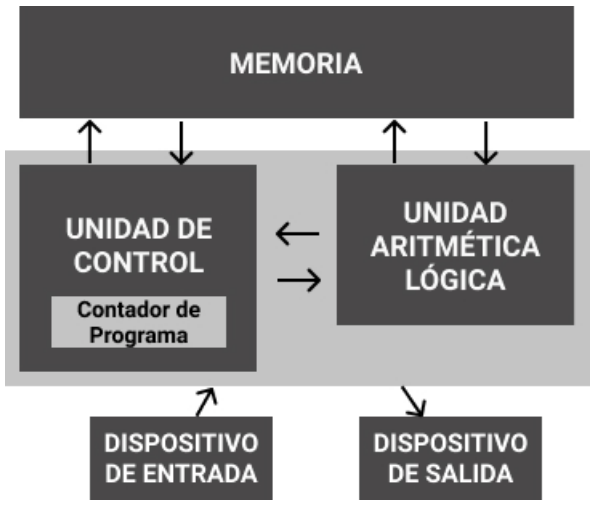
[1] Esto se conoce como **Conocimiento Declarativo** y se compone de afirmaciones de hecho (*statement of facts*).

computers), los cuales poseen una memoria para **guardar secuencias de instrucciones** las que, después, son ejecutadas por un **intérprete**. Es decir, el conjunto de pasos ya no viene fijo a la máquina sino que uno se lo da, de manera que es posible modificar su funcionalidad sin necesidad de cambiar su circuito.

La base de los computadores que usamos en la actualidad proviene, en términos generales, de los de programa fijo y los de programa almacenado. Veamos ahora la arquitectura básica de este último.

3.1 Arquitectura básica de un Computador de Programa Almacenado.

Un computador de programa almacenado suele estar construido a partir de la siguiente arquitectura básica[2]:



donde:

- **Memoria**: Almacena datos y secuencia de instrucciones (también interpretados como datos).
- **Unidad Aritmética Lógica**: Realiza operaciones aritméticas y a nivel de bits (*bitwise operations*).
- **Unidad de Control**: Contiene al **Contador de Programa**, que indica el lugar en la memoria de la siguiente instrucción a ser ejecutada.
- **Dispositivos de Entrada y Salida**: También conocidos como **periféricos**, son aquellos con los que nos comunicamos con un computador (e.g, el teclado [entrada], el monitor [salida], etc).

[2]Se conoce también como Arquitectura de Von Neumann.

En particular, el contador de programa toma la ubicación en la memoria de la primera instrucción. Luego, ésta se ejecuta en la unidad aritmética lógica y, al terminar, vuelve a la memoria. En ese momento, el contador asciende en una unidad (i.e, +1) y continúa con la siguiente instrucción. Al finalizar la secuencia completa, se obtiene una respuesta en el dispositivo de salida correspondiente.

El contador de programa opera de forma lineal o secuencial. No obstante, recordemos que algunas instrucciones pueden contener pruebas, lo que determina el lugar de la memoria al cual apuntará, según si se cumple o no la condición.

Lo relevante del computador de programa almacenado, es que la secuencia de instrucciones parte siendo dato en la memoria y lo es otra vez al terminar su ejecución, lo que nos permite modificarlo como y cuándo queramos.

Todas las instrucciones que creamos en un computador de programa almacenado se hacen sobre la base de **instrucciones primitivas predefinidas**, las cuales son:

- Operaciones aritméticas y lógicas.
- Evaluar pruebas.
- Mover y copiar datos de un lugar de la memoria a otro.

Por otra parte, las instrucciones son ejecutadas por un programa especial llamado **Intérprete**, el cual usa pruebas para cambiar el **flujo de control** y se detiene cuando ya se realizaron todas.

4. Programa Computacional.

Un **algoritmo**, o secuencia de instrucciones, se implementa en un computador a partir de un **Programa Computacional**, el cual se escribe usando un **Lenguaje de Programación**.

En 1936, el matemático inglés **Alan Turing** demostró que es posible programar (*computing*) cualquier secuencia de pasos a partir de **seis instrucciones primitivas básicas**: mover a la izquierda, a la derecha, leer, escribir, escanear y parar/hacer nada.

En la actualidad podemos escribir programas usando más operaciones primitivas de las señaladas por Turing. No obstante, su planteamiento sigue sosteniéndose en el hecho de que es posible escribir una misma serie de instrucciones en varios lenguajes a la vez[3]. A ésto se lo conoce como **Completitud de Turing**.

[3]La diferencia se da en que, en ocasiones, un programa es más fácil escribirlo en un lenguaje que en otro, pero la posibilidad se mantiene en ambos.

4.1 Entendiendo los Lenguajes de Programación.

En este curso trabajaremos con el lenguaje de programación **Python**, pero lo usaremos más de forma práctica para entender lo que iremos estudiando en las clases.

En general, todo lenguaje de programación se constituye de:

- **Constructos Primitivos**: Elementos más simples disponibles en un lenguaje.
- **Sintaxis**: Define si una expresión construida de constructos primitivos está bien formulada o no.
- **Semántica Estática**: Evalúa si una expresión con sintaxis válida tiene sentido.
- **Semántica Completa**: La interpretación de la expresión que es sintácticamente correcta y no tiene errores de semántica estática.

Los constructos primitivos en Python son los tipos de datos como los numéricos (`int`, `float` y `complex`) y cadenas de caracteres (`str`), por mencionar algunos; y los tipos de operadores (aritméticos, de asignación, de comparación, etc).

A partir de los constructos primitivos podemos escribir **expresiones**. Éstos son una combinación compleja pero sintácticamente legal de los primeros, los cuales regresan un nuevo valor.

En Python (y probablemente en cualquier otro lenguaje de programación), la siguiente expresión es sintácticamente válida:

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

Sin embargo, si reemplazamos el operador de adición `+` con un espacio en blanco, obtenemos un **error de sintaxis**.

```
>>> 3 2
  File "<stdin>", line 1
    3 2
      ^
SyntaxError: invalid syntax
```

La **sintaxis** en un lenguaje de programación podemos entenderlo como el **conjunto de reglas** sobre la **forma** de escribir expresiones. El ejemplo de arriba entrega una error porque está determinado en Python que no podemos escribir dos números con un espacio entre ellos.

Es un tanto similar a la sintaxis del lenguaje con el que nos comunicamos. La oración “perro cama plato” es sintácticamente inválida porque, en el idioma español, debe consistir de

sustantivos y al menos un verbo. Un ejemplo válido sería, más bien, “perro dormir plato”.

También es posible que escribamos una instrucción cuya sintaxis sea válida, pero de igual el lenguaje nos regrese un error porque para éste no tiene sentido, como el que vemos a continuación:

```
>>> 3+"hola"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

La sintaxis de la expresión `3+"hola"` es válida, pero para Python la adición entre un valor de tipo numérico (`int` o entero) y uno de cadena de caracteres (`str`), es incompatible. Esto se conoce como un **error de semántica estática**.

Usando el español, un ejemplo de error de semántica estática sería la oración “tú estoy durmiendo”. Su sintaxis es correcta, pero no tiene sentido el “estoy”, ya que está en segunda persona. Lo correcto sería escribirlo como “tú estás durmiendo”.

Python siempre nos señalará y de forma explícita si cometimos un error de sintaxis. En los de semántica estática, como vimos, será un tanto más implícito, pero **nunca lo hará** cuando es un **error de semántica completa**.

Un **error de semántica completa** es aquel donde se obtiene un **resultado que no era el de interés** a pesar de que la expresión es sintácticamente válida y su semántica estática hace sentido. Por ejemplo, cuando un programa se bloquea (*crashes*), entra en un búcle (*loop*) infinito o, simplemente, cuando la respuesta es distinta a la esperada.

Recordemos que un computador ejecuta de forma literal lo que uno le instruye. Por lo tanto, cuando nos enfrentamos a un error de semántica completa tenemos que revisar el código fuente para identificar el problema y depurarlo (*debuging*).

5. Python.

Un programa escrito en Python corresponde a una secuencia de **definiciones** que son evaluadas y **comandos** que instruyen lo que se debe hacer. Estos últimos son ejecutados por el intérprete del lenguaje en una **shell**.

Es posible escribir los comandos de Python directamente en la shell o por medio de un archivo de texto, conocido como **script** y que llevan la extensión `.py`.

Por ejemplo, el siguiente programa llamado `saludo.py` consiste del código que vemos a continuación:

```
#!/usr/bin/env python3
print("¡Hola! :)")
```

Para ejecutar `saludo.py`, usamos el comando:

```
python saludo.py
```

Obteniendo como respuesta:

```
¡Hola! :)
```

En este ejemplo usamos la función `print()` para mostrar el resultado en la consola. Si no la utilizamos **al trabajar con un script**, no aparecerá nada. Esto no es necesario cuando lo hacemos directamente desde la shell, pero esta aplicación no es la mejor opción cuando estamos creando programas más complejos.

5.1 Objetos.

La programación en Python se realiza, principalmente, manipulando **Objetos** (*objects*). Cada uno de ellos tiene definido un **tipo** (*type*), el cual determina las operaciones que podemos hacer con ellos.

Los objetos se dividen en dos tipos:

- **Escalares**: Son los **objetos más básicos** del lenguaje y, por tanto, no tienen subdivisiones. Pueden contener un único valor y es posible usarlos en cualquier tarea en Python.
- **No escalares**: Este tipo de objetos son **divisibles** porque **tienen una estructura interna** y, por consiguiente, podemos acceder a sus elementos.

Los objetos escalares y no escalares tienen subdivisiones entre sí y es posible verificar aquellos tipos usando la función `type()`.

Dentro de los **objetos escalares** es posible encontrar algunos de los siguientes tipos:

- `int`: Son objetos numéricos que representan a **números enteros**.
- `float`: Corresponden a valores de **punto flotante**, los cuales son una aproximación científica a los **números reales**. Son todos aquellos con un `.` entre los dígitos.
- `bool`: Representan los **valores booleanos** `True` y `False`.

- `NoneType`: Son objetos escalares cuyo valor es `None`, que expresan la **ausencia de un tipo** de escalar.

Cada objeto escalar tiene una función para convertir el valor de un tipo a otro. Por ejemplo, pasar el número `3` de tipo `int` a `3.0`, que es de tipo `float`.

```
>>> type(3)
<class 'int'>
>>> float(3)
3.0
>>> type(float(3))
<class 'float'>
```

5.2 Expresiones y Operadores en Python.

Anteriormente estudiamos que las expresiones son una combinación compleja de constructos primitivos. En Python se crean a partir de **operadores** y **objetos**, las que regresan un valor de un tipo determinado.

Sean `i` y `j` dos objetos escalares numéricos. Si son de tipo `int` y/o `float`, podemos realizar las siguientes **operaciones**:

- **Adición** → `i + j`.
- **Sustracción** → `i - j`.
- **Producto** → `i * j`.

En estas tres operaciones, si los dos objetos son `int`, el resultado es `int`. Si al menos uno de ellos es `float`, regresa un valor de tipo `float`.

También podemos dividir:

- **División** → `i/j`.

En esta operación, **el resultado siempre será un objeto de tipo** `float`.

```
>>> 2/1
2.0
>>> type(2/1)
<class 'float'>
```

Otras operaciones que podemos hacer con `i` y `j` son:

- **Resto de la división** → `i % j`.

- **Potencia** → `i ** j` (`i` elevado a `j`).

Las operaciones aritméticas en Python siguen las **reglas de precedencia** que se usa en matemática.

5.3 Variables.

Al programar, muchas veces vamos a querer guardar un resultado en una especie de contenedor informático para acceder a él posteriormente. Éstos reciben el nombre de **Variable** y es un **nombre simbólico asignado a un objeto**, el cual se almacena en la memoria de un computador.

Por ejemplo, digamos que calculamos el siguiente promedio:

```
>>> (3+2+5)/3
3.3333333333333335
```

Para acceder a él podemos copiar, pegarlo en la consola o adentro de la función `print()` y ejecutarlo las veces que queramos. Pero si estamos creando un programa más complejo, escribir aquella línea de código probablemente lo hará más ilegible o, si otra persona lo lee, no siempre sabrá qué es. En ese caso, lo mejor sería **vincularlo** (*binding*) a una variable.

Llamemos a la variable `promedio_1` y, para vincularlo a ese valor, usamos el **operador de asignación** `=`.

```
>>> promedio_1 = (3+2+5)/3
```

Como vemos, con el operador de asignación vinculamos el nombre de la variable `promedio_1` al valor de la expresión `(3+2+5)/3` y siempre deben ser escritos de izquierda (nombre) a derecha (objeto).

Para acceder a la variable, solo tenemos que escribir su nombre.

```
>>> promedio_1
3.3333333333333335
```

Una **buena práctica** al trabajar con variables, es que su **nombre siempre debe representar lo que se está vinculando**. Fácilmente podríamos haber escrito `a = (3+2+5)/3`, pero es posible que, con el pasar del tiempo, no recordemos qué es lo que calculamos en `a`.

También es posible **asignar variables usando otras**, como lo vemos a continuación:

```
>>> suma_2 = 1+15+7
>>> valor_n = 3
```

```
>>> promedio_2 = suma_2/valor_n
>>> promedio_2
7.666666666666667
```

Otra tarea que podemos hacer con variables, es cambiar el valor vinculado a ella. Por ejemplo:

```
>>> promedio_1
3.3333333333333335
>>> promedio_1 = promedio_2 + 1
>>> promedio_1
8.666666666666668
```

Este cambio ocurre para siempre. Es decir, no podemos volver al objeto anterior vinculado a `promedio_1` salvo que lo reasignemos a su expresión original.