# Programación 1

Tecnicatura Universitaria en Inteligencia Artificial

2022

# Apunte 2

## 1. Variables

### 1.1. Declaración de variables

Una **variable** es un objeto de memoria cuyo valor puede cambiar durante el desarrollo del algoritmo o ejecución del programa. La declaración de las variables implica darles un lugar en memoria, un nombre y tipo de dato asociado.

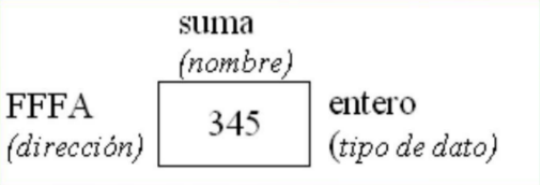


Figura 1: Atributos de una variable

Una variable es un nombre que refiere a un valor. En Python la declaración de las variables se realiza implícitamente a través de la asignación. El operador de asignación nos permite almacenar un valor o resultado en una variable

```
x = 5
```

### 1.2. Nomenclatura

Es fuertemente recomendable utilizar nombres descriptivos para las variables.

Los nombres de variables:

- Pueden ser tan largos como se desee
- Pueden tener letras y números
- Pero NO pueden comenzar con un número

- Es válido usar mayúsculas pero es conveniente utilizar solo minúsculas
- Se puede utilizar el guion bajo ( _ )

Las variables también nos permiten ordenar nuestras ideas en el código, dándoles significado a los valores o a los resultados de expresiones, siendo que para nosotros estos tienen importancia.

```
personas_que_son_multitud = 3
saludo_formal = "Hola"
quiero_dormir = True
```

Estas variables guardan los valores que representan o responden a la idea que queremos expresar. Y para luego obtener los valores, podemos llamar las variables por su nombre.

```
print(personas_que_son_multitud) # 3
print(saludo_formal) # 'Hola'
print(quiero_dormir) # True
```

Nótese que, si bien el comportamiento es el mismo, no es la misma **idea** imprimir simplemente "Hola", que imprimir una variable que represente un saludo formal. Pues el día que nuestra forma de saludar cambie, lo correcto sería expresar que nuestra variable que representa esta noción cambie, y no un valor que se encontraba dentro de un print por alguna razón.

## 1.3. Scope

Luego de que a una variable se le asigna un valor por primera vez se le dice declarada o definida. Y además se dice que comienza su *scope*. El *scope* o alcance de las variables es toda la sección del código donde las mismas se pueden usar. Antes de ser declaradas o luego de ser destruidas las variables no están definidas o se dice que están *fuera de scope*.

Con el operador `del` podemos borrar una varialbe de nuestro programa, para que deje de poder ser utilizada

```
"""
En esta sección de código variable_declarada no fue declarada,
está fuera de scope
"""

variable_declarada = 1

"""
En esta sección de código puedo usar variable_declarada,
está en scope y valdrá 1
"""

del variable_declarada

"""
En esta sección de código variable_declarada fue borrada,
está fuera de scope
"""
```

Por supuesto nada me impide redeclararla luego, iniciando un segundo scope donde la variable vuelve a estar activa con algún valor asignado.

# 2. Tipos

Tenemos una noción de qué significa categorizar cosas, decimos que tal cosa es una manzana y otra cosa es una pera. Existen muchas manzanas y muchas peras, pero entre todas hay algo que es cierto, ninguna manzana es comparable con ninguna pera. Al menos no directamente, sí sabemos que por lo menos ambas categorías son frutas.

En Python, y en la programación en general, también contamos con categorías. A estas categorías les decimos **tipos**. Podemos pensar a los tipos como en el ejemplo anterior, agrupaciones de cosas que sean similares, que tenga sentido comparar. Decimos que un valor `V` es de tipo `T` o también abreviamos que `V` es un `T`.

En Python los tipos básicos se dividen en:

- **Números**:
  - **int** (entero)
  - **float** (de punto flotante)
  - **complex** (complejo)
- **Cadenas de texto** (**str**)
- **Valores booleanos** (**bool**)

Según el tipo de dato, el microprocesador reserva una porción de memoria principal para almacenar los valores que adoptará la variable.

## 2.1. Números

### 2.1.1. Enteros

En Python, los números enteros son todos aquellos números positivos, negativos y el 0, sin parte fraccional. Son exactos y de precisión arbitraria, al contrario de otros lenguajes como C o Java donde no se pueden representar números enteros demasiado grandes. En muchos lenguajes el tipo de dato int es fijo y hay limites a lo que puede almacenar. Si se sobrepasan se produce un desbordamiento u overflow. En Python no existen limites para el entero, si se necesita más memoria se reserva la misma según lo que se necesite (el límite viene dado por la cantidad de memoria disponible).

```
>>> 10 ** 100
>>> -(10 ** 100 - 1) // 3
```

Los números enteros particularmente se podrán representar en otras bases

```
>>> 0b101010 # binario
>>> 0o3472 # octal
>>> 0xabc # hexadecimal
```

### 2.1.2. Flotantes

Los números de punto flotante se llaman así porque se parecen a la notación científica donde nos interesan solo algunos de sus dígitos, lo más importantes, y el exponente del número. Por esta razón los flotantes no son exactos, pero su rango de números representables es enorme. Pueden ir aproximádamente desde $10^-323$ hasta $10^{308}$.

```
# La notación con e significa exponente
# XeY significa X por 10 a la Y
```

```
>>> 1e308
>>> 1e309
>>> 1e-323
>>> 1e-324
```

El rango y manejo del desbordamiento dependen de la arquitectura subyacente.

Puede determinarse ejecutando:

```
import sys
x=sys.float_info.max
print(x)
```

### 2.1.3. Complejos

En Python los números complejos existen integrados directamente en el lenguaje. Un número complejo $a + bi$ se puede representar en Python como `a + bj`. Internamente se utilizan dos números flotantes para representarse, y soportan las operaciones típicas de los complejos que se estudian en matemática.

```
>>> 1j
>>> 1 + 1j
```

La parte que vuelve a un número complejo es la letra $j$. En realidad vuelve a un número completamente imaginario.

## 2.2. Strings

Los strings son un tipo inmutable que permite almacenar secuencias de caracteres. Los strings son sencillamente texto, por lo tanto se separan un poco del resto de los tipos del lenguaje. Aún así Python permite operar con ellos usando la suma, e incluso el producto con un número, algo que no es común en los lenguajes de programación. Existen muchas funcionalidades muy específicas a los strings, que veremos más adelante.

## 2.3. Booleanos

En la vida nos hacemos preguntas que se contestan sí y no. Este concepto se traduce a la programación en la forma de los bools, en honor a George Bool, que desarrolló la forma de operar algebráicamente con estos *valores de verdad*.

```
>>> True
>>> False
```

Se deben escribir con mayúscula al principio pues así lo define el lenguaje. Toda expresión que tenga forma de pregunta, que se pueda responder con verdadero o falso, tendrá un resultado booleano. Como dicho resultado es un valor, podremos operar con él usando operadores lógicos, que se corresponden directamente con la lógica que manejamos en nuestro lenguaje natural en el día a día.

```
>>> True or False
>>> False and True
>>> not False and True
>>> False or not True
>>> 1 < 3 and not 0 == 0.0
```

La última expresión se leería: 1 es menor a 3 y 0 no es igual a 0,0, ¿es esto cierto? ¿sí o no?

### 2.4. NoneType

Por último tenemos un valor especial en Python, que tiene un tipo todo para sí mismo, el NoneType. Es el único valor en ese tipo, para poder diferenciarlo de todo lo demás en el lenguaje, y es None.

Este valor se usar para representar la nada, cuando algo no debería tener un valor. Como la noción de la nada es difícil de expresar, ya que es justamente nada, se creó este valor para representarla.

```
>>> None
```

Los usos de este valor se verán más claros con el uso y la práctica, más adelante.

### 2.5. Conversión entre tipos

La conversión entre tipos es una *reinterpretación* de un valor de un tipo, a un valor del otro, de alguna forma que tenga sentido. Por ejemplo, podemos pensar que 1 y 1.0 son el mismo número. En Python esto no es cierto, pero en la práctica veremos que se pueden usar casi intercambiablemente.

Existen dos tipos de conversiones, implícitas y explícitas. Las implícitas ocurren por sí solas sin que nos demos cuenta, y las explícitas las hemos pedido a propósito.

```
x = 1
print(type(x)) # <class 'int'>
y = x + 1.0
print(type(y)) # <class 'float'>
z = y + 1j
print(type(z)) # <class 'complex'>
```

Efectivamente hemos convertido el 1 a float antes de sumarlo, y luego el 2.0 a complejo antes de sumarlo también.

Python tiene reglas estrictas de cuándo y cómo estas conversiones pueden ocurrir, lo que no significa que no estén disponibles para que usemos algunas de forma explícita, usando el nombre del tipo al que queremos convertir.

```
>>> 1 + "1"
>>> 1 + int("1")
```

La primer línea dará error, a pesar de que la operación es posible. Debemos pedirla explícitamente. Esto Python lo ha decidido así para no tener accidentes donde no entendemos los valores resultantes que hemos obtenido de operaciones complejas entre varios tipos diferentes. El lenguaje de programación JavaScript es famosamente conocido por convertir casi todo por su cuenta, resultando en comportamientos bizarros que se detectan mucho después en la ejecución del programa y no cuando hemos realizado una operación 'no deseada'.

## 3. Operadores

Los operadores están en el medio de expresiones que todavía nos falta resolver.

### 3.1. Aridad

La aridad de un operador es la cantidad de operandos que tiene. Si bien la gran mayoría tiene uno o dos operandos, veremos más adelante que esta idea se puede extender y hablaremos de aridades mayores.

```
# - y + pueden ser unarios o binarios dependiendo el contexto
>>> + 1 - 1
>>> - (+ 1 + 2)
```

## 3.2. Precedencia

Cuando una expresión contiene más de un operador, el orden de evaluación depende del orden de precedencia de las operaciones. Para operadores matemáticos, Python respeta las convenciones matemáticas. La precedencia más alta viene dada por los paréntesis, que pueden ser usados para alterar el orden evaluación según se desee. Las reglas con las siguientes:

- Dado que las expresiones con paréntesis son las primeras evaluadas, el resultado de la expresión `2 * (3 - 1)` es 4 y `(1 + 1)**(5 - 2)` es 8. También pueden usarse paréntesis para facilitar la lectura de una expresión `(minutes * 100) / 60` aunque el resultado no se vea afectado.
- La exponenciación tiene la precedencia más alta, por lo tanto la expresión `1 + 2**3` evalúa a 9, no 27, y `2 * 3**2` evalúa a 18, no 36.
- La multiplicación y división tienen mayor precedencia que la suma y la resta. Por lo tanto, `2*3 -1` evalúa a 5, no 4, y `6 + 4/2` es 8, no 5.
- Los operadores con igual precedencia evalúan de izquierda a derecha (con excepción de la exponenciación).

Se aconseja usar paréntesis en caso de no recordar la precedencia de los operadores de una expresión.

## 3.3. Tipos de operadores

### 3.3.1. Operadores aritméticos

Los operadores aritméticos permiten operar valores numéricos. Como dijimos antes, cuando se mezclan tipos de números, Python convierte todos los operandos al tipo más complejo de entre los tipos de los operandos que forman la expresión. En el Cuadro 1 se presentan los operadores disponibles para operar con números.

| Operador | Descripción | Ejemplo |
|---|---|---|
| `+` | Suma | `r = 3 + 2 # r es 5` |
| `-` | Resta | `r = 4 - 7 # r es -3` |
| `-` | Negación | `r = -7 # r es -7` |
| `*` | Multiplicación | `r = 2 * 6 # r es 12` |
| `**` | Exponente | `r = 2 ** 6 # r es 64` |
| `/` | División | `r = 3.5 / 2 # r es 1.75` |
| `//` | División entero | `r = 3.5 // 2 # r es 1.0` |
| `%` | Módulo | `r = 7% 2 # r es 1` |

Cuadro 1: Operadores aritméticos

### 3.3.2. Funciones y constantes matemáticas

Para poder hacer uso de funciones y constantes matemáticas, debemos importar el módulo `math` (veremos más adelante qué son los módulos). Por ahora para usar las funciones y constantes matemáticas, escribiremos lo siguiente al comienzo del programa:

```
from math import *
```

Algunas funciones y constantes matemáticas del módulo `math` se presentan en los Cuadros 2 y 3.

### 3.3.3. Operadores booleanos

Para operar sobre valores booleanos disponemos de los operadores `and`, `or` y `not`, los llamados operadores lógicos o condicionales. En el Cuadro 4 se brinda una descripción de cada uno de estos operadores.

| Nombre de la función | Descripción |
|---|---|
| `abs(valor)` | Valor absoluto |
| `ceil(valor)` | Redondea para arriba |
| `cos(valor)` | Coseno, en radianes |
| `floor(valor)` | Redondea para abajo |
| `log10(valor)` | Logaritmo, base 10 |
| `max(valor1, valor2)` | Mayor de dos valores |
| `min(valor1, valor2)` | Mínimo de dos valores |
| `round(valor)` | Entero más cercano |
| `sin(valor)` | Seno, en radianes |
| `sqrt(valor)` | Raíz cuadrada |

Cuadro 2: Funciones matemáticas

| Constante | Descripción |
|---|---|
| `e` | 2,7182818... |
| `pi` | 3,1415926... |

Cuadro 3: Constantes matemáticas

| Operador | Descripción | Ejemplo |
|---|---|---|
| `a and b` | El resultado es `True` solamente si `a` es `True` y `b` es True de lo contrario el resultado es `False` | `r = True and False # r es False` |
| `a or b` | El resultado es `True` si `a` es `True` o `b` es `True` (o ambos) de lo contrario el resultado es `False` | `r = True or False # r es True` |
| `not a` | El resultado es `True` si `a` es `False`, de lo contrario el resultado es `False` | `r = not True # r es False` |

Cuadro 4: Operadores booleanos

### 3.3.4. Operadores relacionales

Los valores booleanos son además el resultado de expresiones que utilizan operadores relacionales (comparaciones entre valores). Presentamos los operadores relacionales en la Cuadro 5.

| Operador | Descripción | Ejemplo |
|---|---|---|
| `a == b` | ¿son iguales `a` y `b`? | `r = 5 == 3 # r es False` |
| `a != b` | ¿son distintos `a` y `b`? | `r = 5 != 3 # r es True` |
| `a < b` | ¿es `a` menor a `b`? | `r = 5 < 3 # r es False` |
| `a > b` | ¿es `a` mayor a `b`? | `r = 5 > 3 # r es True` |
| `a <= b` | ¿es `a` menor o igual que `b`? | `r = 5 <= 5 # r es True` |
| `a >= b` | ¿es `a` mayor o igual que `b`? | `r = 5 >= 3 # r es True` |

Cuadro 5: Operadores relacionales

### 3.3.5. Operadores para cadena de caracteres

#### Concatenación

La concatenación nos permite construir una cadena de caracteres a partir de otras dos. Por ejemplo:

```
saludo = "Hola " + "mundo"
print(saludo) # Hola mundo
```

#### Multiplicación

Se puede usar operadora multiplicación para construir una cadena de caracteres en base a otra que se repite. El número indicado en la multiplicación se usará para repetir la cadena tantas veces se indique. Ejemplos:

```
saludo = "Hola"
print(saludo*3) # HolaHolaHola
print(saludo*5) # HolaHolaHolaHolaHola
print(saludo*0) # (en este caso la cadena resultante es la cadena vacía)
```

### Indexación

Los caracteres de una cadena de caracteres están son asociados a un índice, veamos un ejemplo de esto:

```
nombre = "Juan Perez"
```

lo que, internamente, se termina representando como se muestra en la Figura 2

| **Caracter** | **J** | **u** | **a** | **n** | | **P** | **e** | **r** | **e** | **z** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Indice** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Figura 2: Representación de una cadena de caracteres

Podemos probar el siguiente código a continuación:

```
print(nombre, " comienza con ", nombre[0]) #Juan Perez comienza con J
```

### Slicing

Podemos obtener una sub-parte de un string utilizando dos índices, de inicio y fin (sin incluir), respectivamente.

```
saludo = "Hola mundo"
print(saludo[3:8]) # "a mun"
```

Si dejamos en blanco el índice de comienzo, arrancará al inicio de la cadena de caracteres.

```
saludo = "Hola mundo"
print(saludo[:8]) # "Hola mun"
```

Dejando en blanco el índice de fin, seguirá hasta el fin de la cadena.

```
saludo = "Hola mundo"
print(saludo[3:]) # "a mundo"
```

Los strings son un tipo de dato inmutable, sin embargo, podemos crear nuevos strings a partir de otros. Por ejemplo:

```
nuevo_saludo = "Chau" + saludo[4:] # "Chau" + " mundo"
print(nuevo_saludo) # "Chau mundo"
```