

Exactas Programa

Un poco más de Python

Esteban E. Mocskos (emocskos@dc.uba.ar)

Facultad de Ciencias Exactas y Naturales, UBA

CONICET

11/02/2019

Asignación

```
Anastasio = 4  
Pedrito = 8  
Laura = 5  
Micaela = 10
```

Asignación

En `Python` podemos darle nombre a las cosas y asociarles un valor. Esto se llama **asignar** un valor a una **variable**.

`Anastasio` es la **variable** y 4 es el **valor**.

Estado de un programa

El estado de un programa en un momento de su ejecución está definido por el valor de todas sus variables en ese momento. Cuando se analiza **qué** es lo que hace el programa, nos interesa ver y entender **cómo** cambian los valores de las variables.

¿Cómo se ve la ejecución de un programa?

[How do I use this?](#)

Python 3.6

```

1 Anastasio=4
2 Pedrito=8
3 Laura=5
4 Micaela=10

```

[Edit code](#) | [Live programming](#)

⇒ line that has just executed
 ➡ next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

Step 3 of 4

Keep this tool free for everyone by [making a small donation](#)

Help us improve this tool by clicking below whenever you learn something:

I just cleared up a misunderstanding!

I just fixed a bug in my code!

| Frames | Objects |
|--------------|---------|
| Global frame | |
| Anastasio | 4 |
| Pedrito | 8 |

Generate permanent link

Generate shortened link

Click the button above to create a permanent link to your visualization. To report a bug, paste the link along with a brief error description in an email addressed to philip@pgbovine.net

Generate embed code

To embed this visualization in your webpage, click the 'Generate embed code' button above and paste the resulting HTML code into your webpage. Adjust the height and width parameters and change the link to <https://> if needed.

- Este sitio permite ejecutar **paso a paso** nuestro programa.
- Nos permite ver el resultado de cada instrucción (**estado** del programa).

Listas

- En Python existen las **listas**, que sirven para almacenar valores:

```
Anastasio = []  
Pedrito = []  
Laura = []  
Micaela = []
```

```
Anastasio.append(4)  
Pedrito.append(8)  
Laura.append(5)  
Micaela.append(10)
```

```
Anastasio.append(6)  
Pedrito.append(9)  
Laura.append(6)  
Micaela.append(13)
```

- ¿Qué pasa si ejecuto la línea `print("Anastasio:", Anastasio)`?

La salida me muestra: `Anastasio: [4, 6]`

Es una manera *linda* de ver el contenido de la lista... pero hay algo más ahí, ¿no?

Se puede definir una lista (por extensión) como:

```
milistita = [2, -1, 4, -2, 8, 17]
```

Cadenas de caracteres

Definición

Es una secuencia de caracteres definida por medio de comillas, es parecida a una lista, pero no es igual (es un tipo *immutable*):

```
'Hola, trencito'
```

Las operaciones básicas (algunas también funcionan con listas) son:

- **+**: concatenación. `'Hola' + ', trencito'` da `'Hola, trencito'`.
- **int**: convierte una cadena a número entero. `int('33')` da 33.
- **float**: convierte una cadena a número con coma. `float('4.5')` da 4.5.
- Al igual que con cualquier lista, dos de las funciones más usadas son:
 - **[]**: para acceder a los contenidos de posiciones individuales dentro de una cadena. Por ejemplo, `'Hola'[2]` da `'l'`.
 - **len**: devuelve la longitud de la cadena de caracteres. `len('abc')` devuelve 3.
- **lower**: pasa una cadena a minúsculas. `'Hola'.lower()` da `'hola'`.
- **upper**: similar a la anterior, pero pasa a mayúsculas. `'Hola'.upper()` da `'HOLA'`.

Funciones en Python

- Son una construcción que permite *encerrar* un *pedacito* de programa.
- Así como `append`, hay muchísimas funciones que se pueden utilizar y aprovechar.
- Permiten definir cierto comportamiento interesante y no tener que volverlo a programar cada vez.
- Los lenguajes de programación tienen un mecanismo para definir funciones.
- Los valores que recibe una función se llamas **parámetros** o **argumentos**.

Tabulación

Python *sabe* donde termina la definición de una función por la tabulación: las instrucciones que componen la función están un *tab* o 4 espacios hacia la izquierda.

Tabulación, el retorno

Es **importante** usar una cantidad de espacios o tabulación, pero no mezclar, sino empiezan a aparecer errores muy raros de Python. **¡Sean prolijos!**

Ciclos

- El **while** permite repetir una serie de instrucciones mientras se cumpla una condición.
- Si, desde el principio, sabemos que el rango del ciclo es fijo, se puede usar **for**.
- Definamos la función `suma_elem`, que suma todos los elementos de una lista usando **for** y **while**:

Con **while**:

```
def suma_elem(l):  
    suma = 0  
    i = 0  
    while i < len(l):  
        suma = suma + l[i]  
        i = i + 1  
    return suma
```

Con **for**:

```
def suma_elem(l):  
    suma = 0  
    for i in range(0, len(l), 1):  
        suma = suma + l[i]  
    return suma
```

- `range(inf, sup, paso)`: devuelve una estructura que toma los números desde `inf` hasta `sup` de a `paso`. Por default, `inf` vale 0 y `paso` vale 1 (si no se los escribe explícitamente).

Otra estructura de control: `if`

- Permite ejecutar una serie de instrucciones si se cumple cierta condición.
- Supongamos que queremos usar la función `proc_jugadas` cuando la lista de jugadas no esté vacía o si lo estuviera, que el resultado fuera `-1`:

```
if Anastasio!=[]:
    suma_Anastasio = proc_jugadas(Anastasio)
else:
    suma_Anastasio = -1
if Pedrito!=[]:
    suma_Pedrito = proc_jugadas(Pedrito)
else:
    suma_Pedrito = -1
if Laura!=[]:
    suma_Laura = proc_jugadas(Laura)
else:
    suma_Laura = -1
if Micaela!=[]:
    suma_Micaela = proc_jugadas(Micaela)
else:
    suma_Micaela = -1
```

- Los dos puntos (`:`) son obligatorios, ¡No olvidarse!

Comparaciones y condiciones

- Se pueden realizar distintas comparaciones:
 - `<` menor
 - `<=` menor o igual
 - `>` mayor
 - `>=` mayor o igual
 - `==` igual
 - `!=` distinto
- También se pueden combinar distintas condiciones utilizando los operadores lógicos:
 - **not** negación, si se aplica a `True`, da `False` y a la inversa.
 - **and** se usa `x and y`. Solo da `True` cuando `x` e `y` son `True`.
 - **or** se usa `x or y`. Da `True` cuando alguna de las dos (o las dos) es `True`.
- Esto aplica tanto para las condiciones del `if` como a las del `while`.

```
if a>x and a<y:  
    c = x*x+y*y  
else:  
    c = 2*x*y
```

```
if a*a>x or a<y*y:  
    c = a*a-y*y  
else:  
    c = x*y/2
```

Módulos para usar otras funciones

- Si bien `Python` tiene muchas funciones que se pueden usar *directamente*, hay muchas otras que están disponibles como **módulos**.
- Un **módulo** es una colección de funciones que alguien (o una comunidad) desarrollaron y empaquetaron para que estén disponibles para todo el mundo.
- Para que las funciones estén disponibles para ser utilizadas en mi programa, tengo que usar la instrucción `import`.
- Si quiero generar números aleatorios, que están en el módulo `random`, tengo que escribir:

```
import random
```

```
print(random.random())
```

```
print(random.random())
```

```
print(random.random())
```

- ¿Cómo sé que funciones o módulos hay? **!!!Google!!!**

Existe vida más allá del `pythontutor`: `spyder`

- open source
- cross-platform
- integrated development environment (IDE)
- incluye un editor de texto que remarca las palabras clave del lenguaje
- tiene soporte para distintas versiones de `Python`
- permite escribir programas y probarlos de manera muy sencilla
- En las máquinas de los laboratorios, ya está instalado y listo para usarse (tippear `spyder3` o `spyder` como comando).
- Para aquellos que tienen máquina Windows:
`https://www.anaconda.com/distribution` y bajar `Python 3.7 version` para Windows (64 bits para máquinas nuevas, 32 bits si tienes una medio viejita).

Un entorno de desarrollo en Python: spyder

The screenshot displays the Spyder Python IDE interface. The main window is divided into three panes:

- Editor:** Contains a Python script named `temp.py` with the following code:


```
1 # -*- coding: utf-8 -*-
2 """
3 Editor de Spyder
4
5 Este es un archivo temporal
6 """
7
8 Anastasio=[]
9 Pedrito=[]
10 Laura=[]
11 Micaela=[]
12 Anastasio.append(4)
13 Pedrito.append(8)
14 Laura.append(5)
15 Micaela.append(10)
16 Anastasio.append(6)
17 Pedrito.append(9)
18 Laura.append(6)
19 Micaela.append(13)
20 print("Anastasio:", Anastasio)
```
- Variable explorer:** Shows a table of variables defined in the script:

| Name | Type | Size | Value |
|-----------|------|------|----------|
| Anastasio | list | 2 | [4, 6] |
| Laura | list | 2 | [5, 6] |
| Micaela | list | 2 | [10, 13] |
| Pedrito | list | 2 | [8, 9] |
- IPython console:** Displays the output of the script execution:


```
Python 3.4.5 (default, Jan 9 2018, 14:25:36)
Type "copyright", "credits" or "license" for more information.

IPython 5.4.1 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
?quickref -> Quick reference,
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: runfile('/home/esteban/.config/spyder-py3/temp.py', wdir='/home/esteban/.config/spyder-py3')
In [2]: runfile('/home/esteban/.config/spyder-py3/temp.py', wdir='/home/esteban/.config/spyder-py3')
In [3]: print("Anastasio:", Anastasio)
Anastasio: [4, 6]
In [4]:
```

At the bottom of the interface, a status bar shows: Permissions: RW, End-of-lines: LF, Encoding: UTF-8, Line: 17, Column: 11, Memory: 27 %.

Nano Jack: como el Blackjack, pero con problemitas



- Es uno de los juegos que habitualmente se encuentran en los casinos.
- Se juega entre varios jugadores (más de dos).
- Se usan las cartas francesas (las de poker).
- Cada jugador pide cartas tratando de que sus valores *sumen* 21.
- Si te pasas de 21, perdiste.

Nano Jack

Vamos a tomar el *espíritu* del Blackjack para armar un juego que podamos implementar con un programa, así que las reglas van a estar relajadas (ya las vamos a ir viendo).

Empecemos con cartas de poker



- Cada mazo de cartas tiene cuatro palos.
- Dos palos son rojos, dos negros.
- Las cartas van del 1 al 10 y tres figuras: J, Q, K.
- A las figuras se les asigna valores: J vale 11, Q vale 12 y K vale 13.

Pensemos entre todos

Nuestra tarea será hacer un programa de computadora que *simule* varios jugadores en una partida de `Nano Jack`. ¿Cómo encaramos esto?

Modelando el juego

- Cuando se enfrenta una situación así, lo mejor es revisar cómo es el juego y cuáles son sus principales características.
- Si todo fue como lo planeado, tendremos algunos mazos de cartas para jugar un poco entre nosotros.
- Junto con divertirnos y conocernos, tratemos de buscar qué sería lo que un programa tendría que ir haciendo para jugar.

¿Qué esperan?

¡A armar los grupos y a jugar un ratito!

¡A trabajar!

- Con todo lo que vimos, podemos empezar a solucionar el problema planteado en el primer taller.
- Recuerden que los docentes estamos para ayudar, aprovechen a consultar **todo**, no se traben.
- El material de las clases y las actividades del curso van a estar en el campus virtual (esperamos que pronto). El curso depende del Departamento de Computación (para que lo busquen a partir de ahí).
- Si no tiene cuenta, deberían poder acceder como “guest”. El curso depende del *Dto de Computación* y está asociado al cuatrimestre de *Verano 2019*.

Importante

Aprender a programar se basa en **equivocarse** y aprender de los errores, ¡No tengan miedo de experimentar y preguntar!