Primeros pasos en Python

ExactasPrograma

Facultad de Ciencias Exactas y Naturales, UBA

Verano 2019

Objetivos

- Buscamos descubrir que la computadora es una **herramienta** que brinda inmensas posibilidades.
- Vamos a **resolver** problemas usando la computadora.
- Aprender a encarar problemas, plantearlos y resolverlos, en el medio vamos a tener que **programar**.
- Vamos a **amigarnos** con la computadora y empezar a ver que puede ser nuestra aliada aún cuando parezca que nos vamos a dedicar a cosas muy alejadas de la tecnología.

El cuerpo docente está para ayudar, **pregunten**, **experimenten**, **equivóquense** y **vuelvan a preguntar**. ¡Así funciona esto!

Python

- Open Source
- Cross-platform
- Tiene un fuerte desarrollo en una gran variedad de áreas:
  - Desarrollo de aplicaciones con interface gráfica (jueguitos por ejemplo).
  - Análisis de datos y simulación numérica.
  - Desarrollo para Web e Internet.

Es un lenguaje que se usa cada día más.
Aprender Python ***es*** *una inversión segura.*

Modelando un juego de cartas

- Una de las tareas que se hacen cuando se programa es representar algún aspecto del mundo real, se lo llama **modelar**.
- La computadora es una herramienta **fundamental** para esto.
- ¿Cómo hacemos para *modelar* que al jugador "Anastasio" le tocó un cuatro de picas en un juego de cartas? ¿Qué partes nos podrían interesar de este hecho?
- Vamos a suponer que, para lo que nos interesa **modelar** del juego, solo nos importa el valor.
- ¿Cómo hacemos ahora para decir algo de esto en `Python`?
- Mi primer programa en `Python`:

```
Anastasio = 4
```

Asignación

En `Python` podemos darle nombre a las *cosas* y asociarles un valor. A esto se lo llama **asignar** un valor a una **variable**.
Anastasio es la **variable** y `4` es el **valor**.

Más jugadores

- Ahora queremos modelar que al siguiente jugador, “Pedrito”, le tocó un ocho de diamantes... ¿Cómo hacemos?
- Vamos a *alargar* nuestro programa. Vamos a escribir otra **instrucción**:

```
Anastasio = 4
Pedrito = 8
```

- ¿Y con el resto de los jugadores?

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

Ejecución de un programa

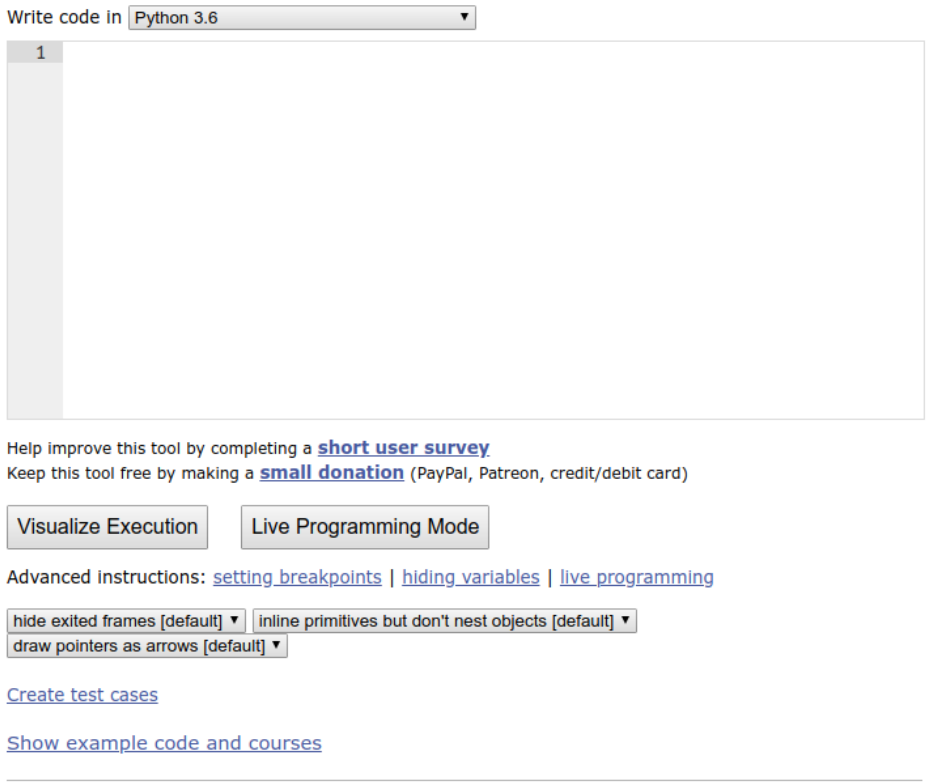
Aunque parezca sencillo, ya tenemos un programa en Python y podríamos *ejecutarlo*. Para hacerlo, necesitamos poder *escribir* el programa y tener instalado un *intérprete* de Python para poder ejecutarlo.

Ejecutando mi primer programa en Python

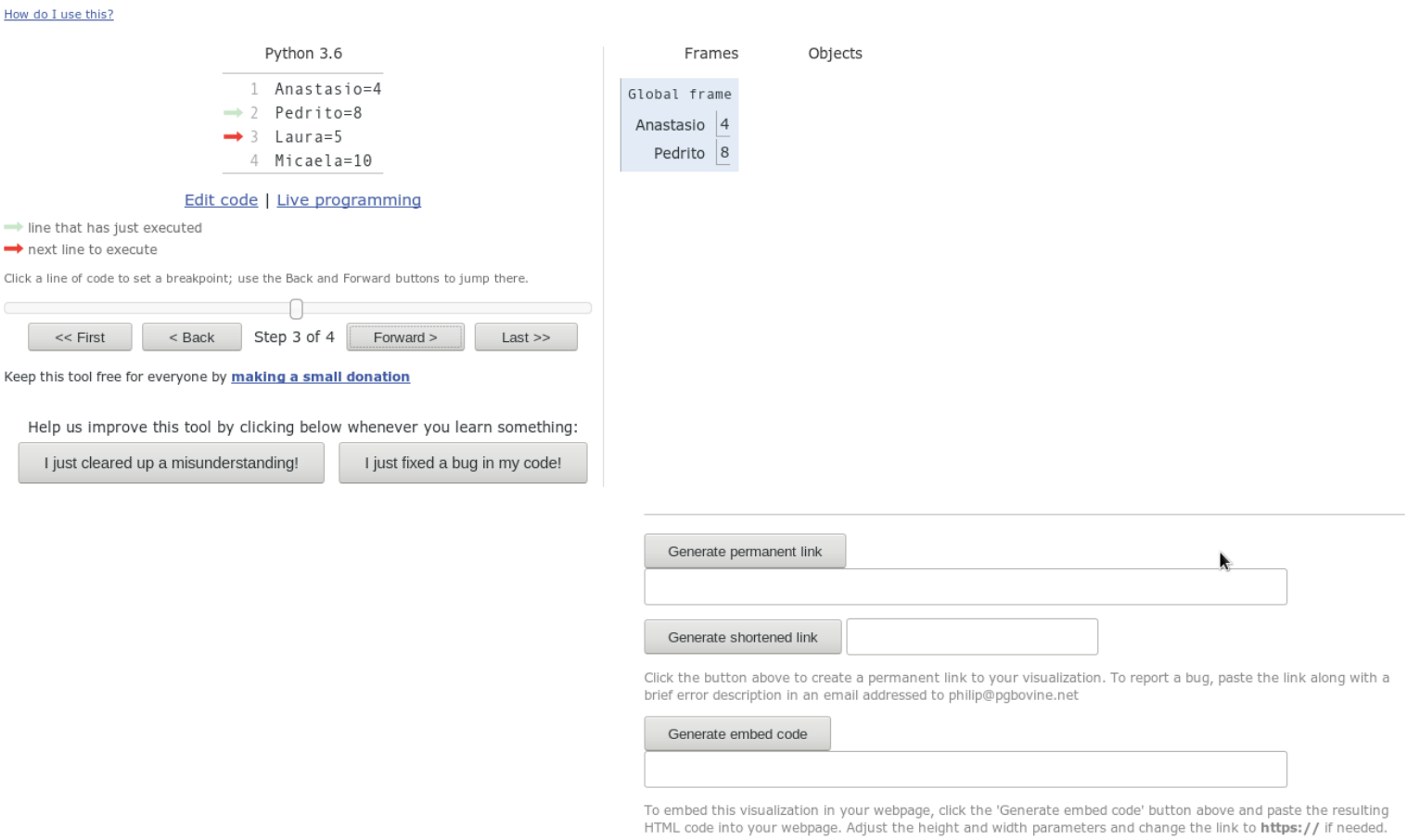
- http://pythontutor.com
- Vamos a usar un sitio para hacer nuestros primeros programas en Python.
- Elegir Visualize your code and get live help now de la pantalla principal.
- Del menú al lado de Write code in, asegurarnos que dice Python 3.6 (ahora es el default).
- Escribir el programa:

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

- Hacer click en Visualize Execution.



¿Cómo se ve la ejecución?



- Este sitio permite ejecutar **paso a paso** nuestro programa.
- Podemos **ver** el resultado de cada instrucción.

¿Cómo hago para modelar una segunda carta?

- Podemos hacer esto para modelar la segunda jugada:

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

Anastasio2 = 6
Pedrito2 = 9
Laura2 = 6
Micaela2 = 13
```

- Pero si ahora queremos simular la tercera, ¿Vamos a definir más variables?

Así no va...

Necesitamos un mecanismo que nos permita guardar las distintas jugadas de cada jugador.

Listas

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

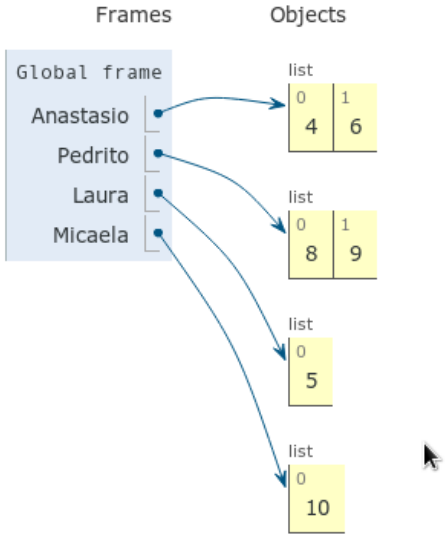
```
Anastasio = []
Pedrito = []
Laura = []
Micaela = []
```

  Empezamos con las listas vacías, no contienen ningún elemento.
- Veamos ahora cómo quedarían las distintas instrucciones:

```
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)
```

¿Y si lo ejecuto?

Escribir el programa anterior y ejecutarlo en `python tutor`.



Una función en `Python`

La función `append` agrega un elemento al final de la lista.

¿Qué vendría a ser un programa?

Vamos a comenzar con esta definición de *entre casa*:

> Un programa es una receta de instrucciones que se siguen desde la primera hasta la última y están escritas de una manera particular.

Hay dos puntos importantes:

- **Receta**: da la idea de una serie de instrucciones que deben seguirse al pie de la letra y que **no** son infinitas.
- **Escritas**: la forma de dar las instrucciones sigue una *sintaxis* o forma de escribir pautada. Es lo que se conoce como **lenguaje de programación**.

Funciones

- Una construcción que permite *encerrar* un *pedacito* de programa. En los libros de recetas, corresponden con procedimientos más sencillos que la receta actual (ejemplo: *bata dos claras a punto nieve*).
- Así como `append`, hay muchísimas funciones que se pueden utilizar.
- Permiten definir cierto comportamiento interesante y no tener que volverlo a escribir cada vez.
- Todos los lenguajes de programación tienen un mecanismo para definir funciones.
- Los valores que recibe una función se llaman **parámetros** o **argumentos**:

```
def elNombreQueSeMeCanta(a, b, c):
    x = (a-b)*c
    y = (a*a - b*b)/c
    r = (x-y)*(x-y)/(x+y)
    return r

ojito = elNombreQueSeMeCanta(2, 4, 7)
print(ojito)
```

Funciones: a probarlo

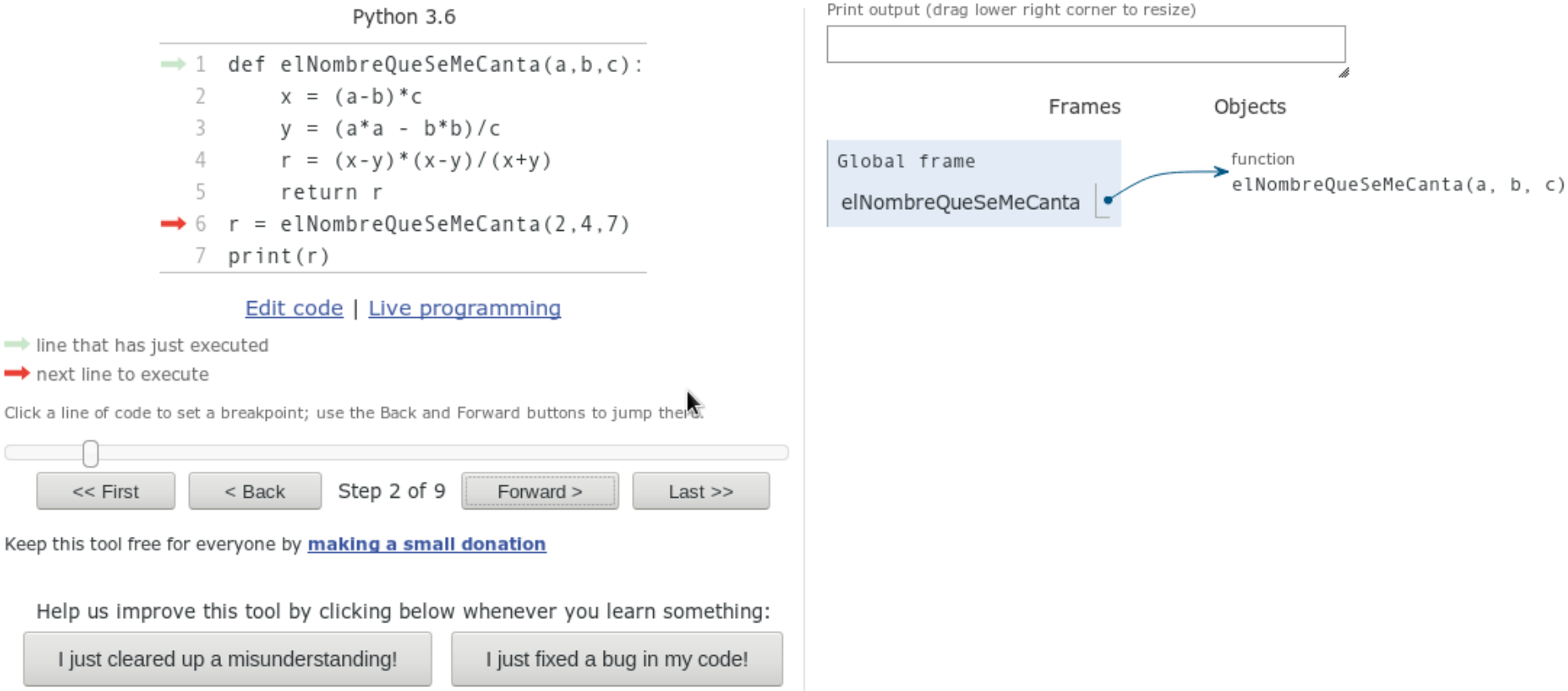
```
def elNombreQueSeMeCanta(a, b, c):
    x = (a-b)*c
    y = (a*a - b*b)/c
    r = (x-y)*(x-y)/(x+y)
    return r

ojito = elNombreQueSeMeCanta(2, 4, 7)
print(ojito)
```

Tabulación

`Python` *sabe* donde termina la definición de una función por la **tabulación**. En el caso de la función `elNombreQueSeMeCanta` las instrucciones que componen la función están un *tab* hacia la izquierda.

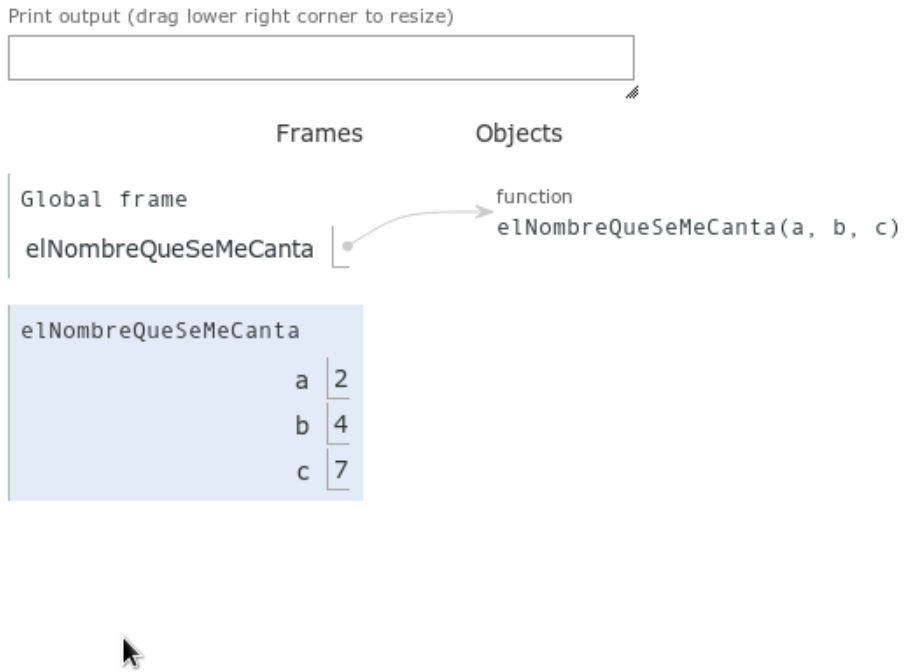
Funciones: ¿Si lo ejecuto?

Vamos a escribir algo parecido al programa anterior y a ejecutarlo.



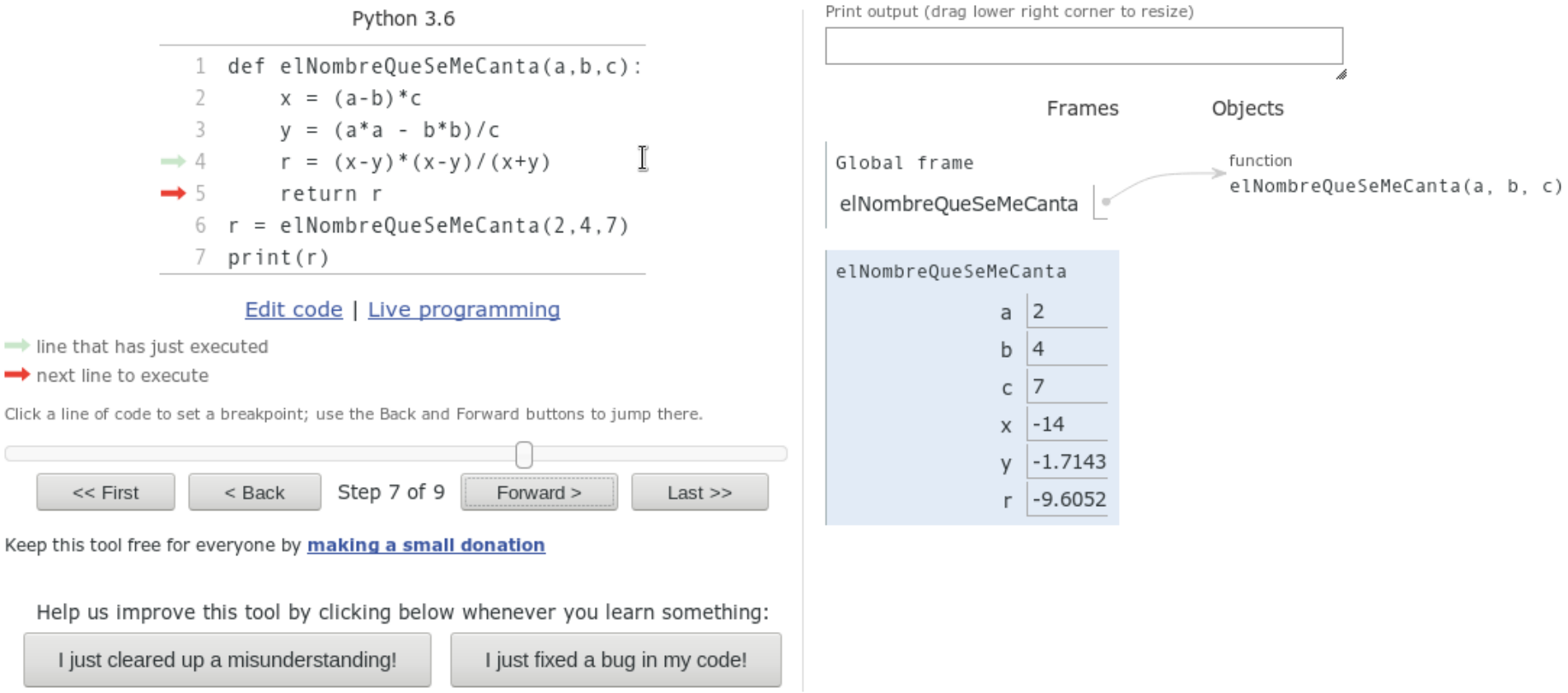
A esta altura de la ejecución del programa, `Python` *sabe* que existe una función llamada `elNombreQueSeMeCanta` que recibe tres argumentos.

¡Sigamos ejecutando!



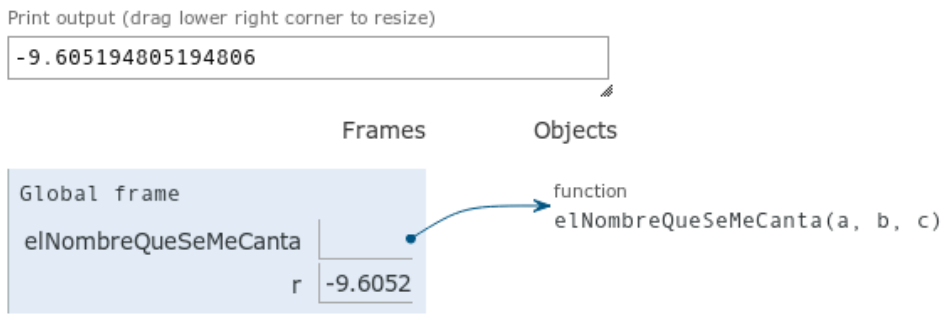
En este paso, `Python` necesita obtener el resultado de la función `elNombreQueSeMeCanta` con los argumentos 2,4 y 7.

¡Sigamos ejecutando!



Acá, `Python` está a punto de devolver el **resultado** de la ejecución de la función. La instrucción `return` le indica a `Python` que el valor de la expresión que sigue es el resultado de la función.

Y la ejecución termina así...



El valor que *devolvió* la función `elNombreQueSeMeCanta` fue asignada a una variable `r`, que si bien se llama igual que la definida dentro de la función... **¡es otra!**

Las variables que se definen dentro de una función, solo **existen** dentro de esa función. A esto se lo conoce como **alcance** de una definición.

Strings: listas raras de caracteres

- Vimos algunos ejemplos de listas con números enteros. Veamos ahora que podemos definir también estructuras con caracteres.
- Para definir un *string* (cadena de caracteres), podemos hacer esto:

```
nombre = "Marianito"
```

  Esto define la variable `nombre` que contiene una cadena de caracteres.
- Podemos imprimir el string:

```
print(nombre)
```

- Pero podemos hacerlo mucho más divertido:

```
print("Mi nombre es:", nombre)
```

  El resultado de esto es que, en pantalla, vamos a ver el contenido de dos cadenas... ¿Cuáles? ¡Probémoslo!
- Una vez que lo probemos y si no cambiamos nada, lo que salió en pantalla fue:
  `Mi nombre es:Marianito`
  Se ve medio feo, ¿no? Está pegoteado el nombre y ":". ¿Cómo lo arreglamos?

Metiéndose con las cadenas

Hay algunas operaciones interesantes para hacer con cadenas (también con listas en general):

- `len(l)`: es una función que recibe una cadena de caracteres (o una lista) y te devuelve la cantidad de elementos que tiene (viene de "length", longitud). `len([])` da...**¡0!**
- `[]`: es un *operador* para acceder a cualquier elemento de una cadena o de una lista:

```
nombre = "Marianito"
print(nombre[0])
print(nombre[5])
```

  Hay que tener cuidado con acceder a los elementos fuera de rango, aparecen errores feos.
- También se pueden crear listas usando `[]`: `[1,2,3,10,99]`
- `+`: concatenación, toma dos strings (o listas) y los *pega*:

```
nombre = "Marianito"
apellido = "Cachirulo"
print("Nombre completo: ", apellido, ", ", nombre)
print("Nombre completo: ", apellido + ", " + nombre)
```

  A probarlo y ver la diferencia.

Los strings no son listas, son parecidos

- Si bien los strings son *casi* como listas, son **inmutables**.
- Esto significa que, una vez definidos, no se pueden cambiar:

```
nombre = "Marianito"
nombre[8] = 'a'
```

Esto **no** funciona da una error que dice
`TypeError: 'str' object does not support item assignment.`
Significa que estamos queriendo cambiar algo que **no** se puede cambiar.

- La manera para trabajar esto es usando una lista de verdad:

```
nombre = "Marianito"
milista = list(nombre)
milista[8] = 'a'
milista[len(milista)-1] = 'a' #Alternativa
print(milista)
print(nombre)
```

Ciclo, triciclo, ciclotímico: ciclo que te ciclo y no reciclo

- Quiero imprimir en pantalla los números del 1 al 10:

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
```

¡Qué gran programa!

- Bueno, bueno, dejen de tirar cosas y abuchear, lo podemos hacer así:

```
i = 1
while i<=10:
    print(i)
    i = i+1
```

- Escribámoslo en el tutor y veamos cómo funciona.

Otra estructura de control: if

- Permite ejecutar una serie de instrucciones si se cumple cierta condición.
- Supongamos que queremos cambiar un valor dependiendo si vale o no 2:

```
mivalor = 2
if mivalor==2:
    mivalor = 5
else:
    mivalor = 8

print(mivalor)
```

Los dos puntos (:) son obligatorios,

¡No olvidarse!

- Veamos un ejemplo más divertido:

```
def multiploRaro(a):
    if a==2:
        res = a*a+1
    elif a==3:
        res = a+a-1
    elif a==7:
        res = a*a+5
    elif a==11:
        res = a*(a-2)+9
    else:
        res = a*a-2
    return a
```

Y ahora a probarlo:

```
prueba=multiploRaro(8)
print(prueba)
```

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
```

Ejercicios

1. Defina una función que tome dos cadenas de caracteres como parámetro y devuelva la de mayor longitud. Complete el siguiente programa:

```
def mas_larga(l1,l2):
    if ______:
        return ___
    else___
        return ___
nombre1="Pepe"
nombre2="Chirizo"
res=mas_larga_____
print(res)
```

2. Defina una función que recibe una lista y devuelve la cantidad de letras `e` que contiene.

```
def cant_e ____:
    i=0
    count=0
    while i<____
        if ___[i]=='e'____
            count=count___
        i____1
    return _____
```

3. Defina una función que tome una lista y cambie todas las vocales por `-`.
4. Defina la función `mezclar` que tome dos listas y devuelva una lista que sea el resultado de intercalas elemento a elemento. Por ejemplo: si intercalamos `Pepe` con `Jose` daría `PJeopsee`.

Links útiles

`https://campus.exactas.uba.ar`