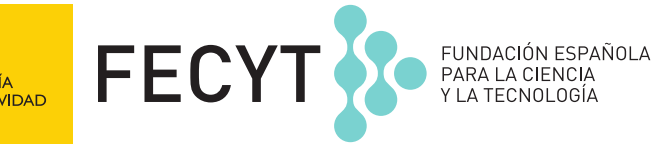
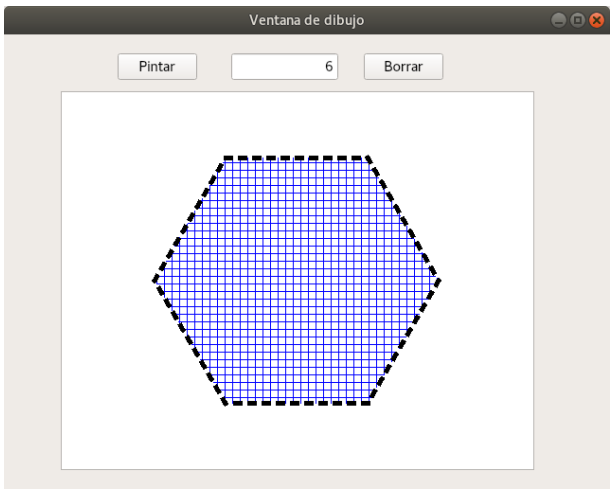
Aplicaciones de la programación matemática con Python: módulo turtle y PyQt5.

Luis Cabello & Paco Villegas

ESTALMAT - ANDALUCÍA





Índice

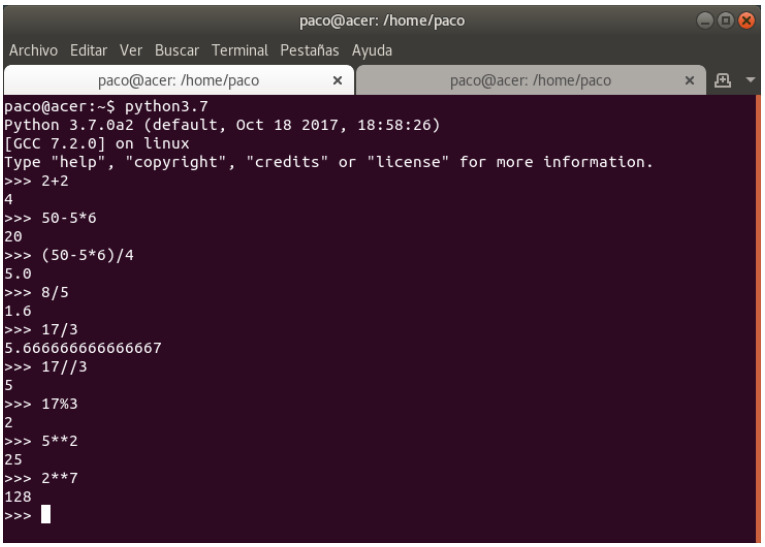
1. Introducción al lenguaje de programación Python

"Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.
Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.
Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License, que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores."
`https://es.wikipedia.org/wiki/Python`
Para esta sesión, nosotros no solo usaremos Python, también haremos uso de PyQt. PyQt es una adaptación de la biblioteca gráfica Qt[a] para el lenguaje de programación Python. Con ella podemos desarrollar interfaces gráficas de usuario usando además, el lenguaje Phyton.

[a] Qt es una biblioteca de interfaz gráfica de usuario multiplataforma muy potente y popular.

En la primera sesión ya visteis cómo trabajar en modo interactivo, en el cual se escriben las instrucciones en un intérprete de comandos, pudiendo ver el resultado de su evaluación inmediatamente. Esto nos permite probar el código antes de usarlo como parte de un programa.

Ejemplo del modo interactivo:



Para salir del modo interactivo usaremos `quit()` o ctrl + D.

1.1. El IDE de Python Eric6 y QT Designer

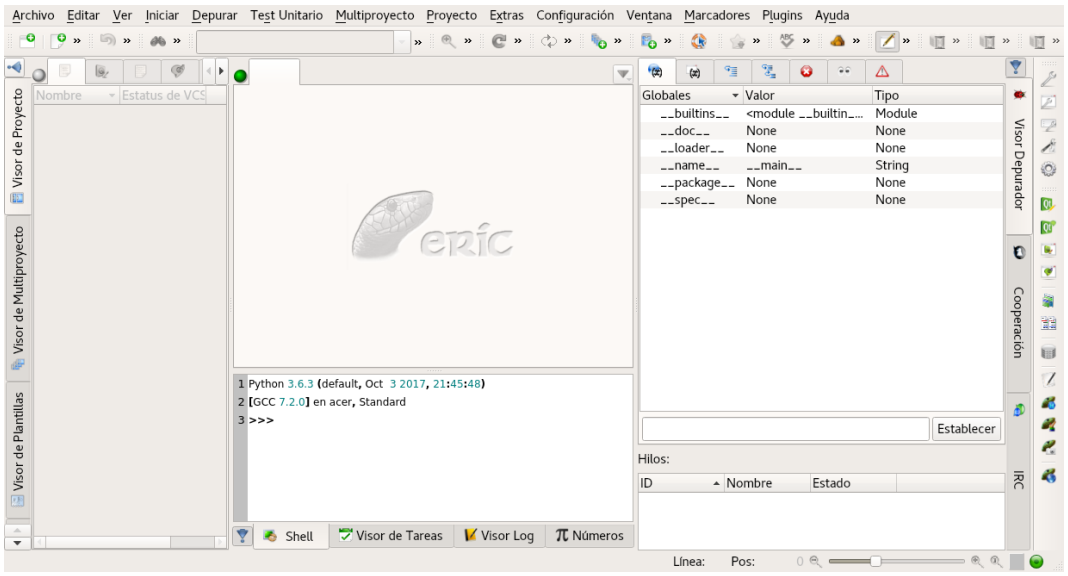
Eric6 es un completo IDE/editor para Python y Ruby, escrito en Python. La lista de características que integra es muy numerosa y nosotros solo usaremos algunas de ellas.

Para instalar ambos programas (Eric6 y QT Designer) en distribuciones basadas en el sistema de paquetes .deb solo tenemos que escribir, como administrador del sistema[1]:

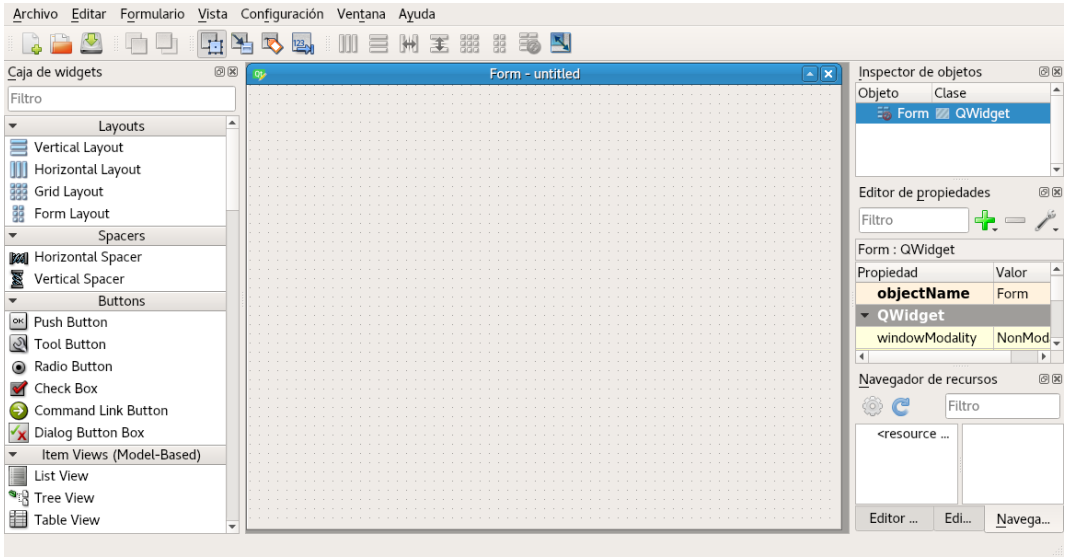
```
# apt-get install eric pyqt5-dev-tools qttools5-dev-tools
```

Si lo ejecutamos, y tras aceptar las primeras ventanas de configuración, veremos una pantalla similar a la captura:

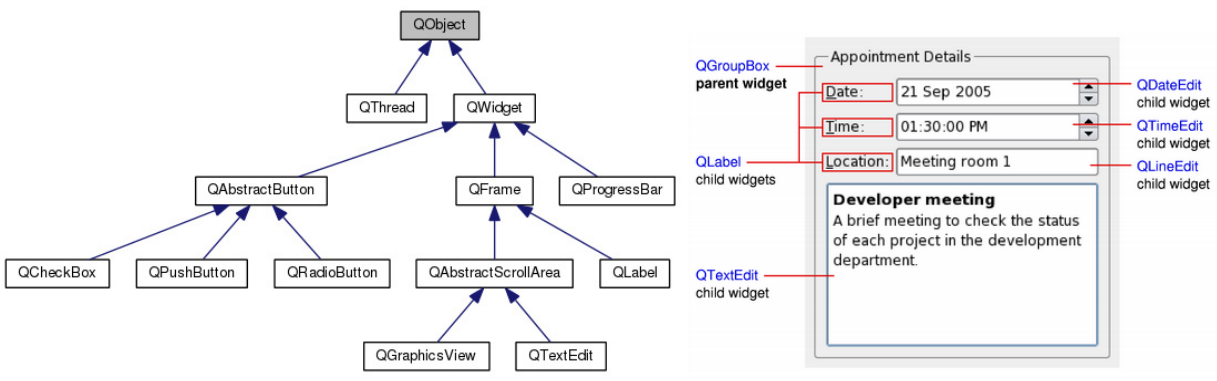
[1] Para instalarlo en Windows u otras plataformas revisar `https://eric-ide.python-projects.org/eric-download.html`



Permite (al integrar Qt Designer) desarrollar aplicaciones gráficas o de consola. Hace posible el desarrollo de aplicaciones gráficas complicadas muy rápidamente. Haciendo uso de Qt Designer podemos diseñar las ventanas de forma gráfica. En el desarrollo de la sesión analizaremos algunas de las opciones de ambos programas.



La programación de una interfaz gráfica se basa en el paradigma de POO (Programación Orientada a Objetos). Cada elemento usado para crear nuestra interfaz es un objeto que tiene sus atributos y sus métodos definidos por la clase de la cual es la instancia.



1.2. Cuestiones previas sobre Python

1.2.1. Comentarios

Los comentarios en Python se pueden poner de dos formas:
Escribiendo el símbolo almohadilla (#) delante de la línea de texto donde está el comentario.
Escribiendo triple comilla doble (‘‘’’’) al principio y al final del comentario (que puede ocupar más de una línea).

```
'''
Comentario más largo que ocupa varias
        líneas en Python
'''
#Comentario de una sola línea
print ("Hola mundo") # Podemos añadir comentarios al final de una línea de código
```

¿Qué saldría en pantalla?

1.2.2. Variables

Variable es una porción de memoria del ordenador que se utiliza en Python para guardar un dato (información) y a la que se le asigna un identificador. **Ejemplo:**

```
radio=4
radio="Texto"
print(radio)
```

¿Qué saldría en pantalla?

1.2.3. Tipos de Datos en Python

Los tipos de datos, que se pueden asignar a las variables, soportados por Python, son los siguientes (fuente `https://es.wikipedia.org/wiki/Python`)

Tipo	Clase	Notas	Ejemplo
str	Cadena	Inmutable	’Cadena’
unicode	Cadena	Versión Unicode de str	u’Cadena’
list	Secuencia	Mutable, puede contener objetos de diversos tipos	[4.0, ’Cadena’, True]
tuple	Secuencia	Inmutable, puede contener objetos de diversos tipos	(4.0, ’Cadena’, True)
set	Conjunto	Mutable, sin orden, no contiene duplicados	set([4.0, ’Cadena’, True])
frozenset	Conjunto	Inmutable, sin orden, no contiene duplicados	frozenset([4.0, ’Cadena’, True])
dict	Mapping	Grupo de pares clave:valor	{’key1’: 1.0, ’key2’: False}
int	Número entero	Precisión fija, convertido en long en caso de overflow.	42
long	Número entero	Precisión arbitraria	42L ó 456966786151987643L
float	Número decimal	Coma flotante de doble precisión	3.1415927
complex	Número complejo	Parte real y parte imaginaria j.	(4.5 + 3j)
bool	Booleano	Valor booleano verdadero o falso	True o False

- Mutable: si su contenido (o dicho valor) puede cambiarse en tiempo de ejecución.
- Inmutable: si su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución.

Conversión de datos en Python Solo vamos a analizar aquellos que usaremos en la sesión:

int(expresión) convierte la expresión en un número de tipo entero.

str(expresión) convierte la expresión en una cadena de texto.

float(expresion) convierte la expresión en un número decimal.

```
>>> int('3')
3
>>> str(3)
'3'
>>> float(3)
3.0
```

1.2.4. Operadores Matemáticos en Python

Además de los operadores matemáticos (+,-,*,/) ya conocidos en casi todos los lenguajes de programación, citamos los siguientes:

** es el operador de potencia. **Ejemplo:** 4**3=64

Para la división hay dos operadores adicionales, // y %, que devuelven, respectivamente, la parte entera del resultado de la división y el resto.

Ejemplo:
```
numero = 9 // 2
numero = 9% 4
```

¿Qué hay en numero después de la primera asignación? ¿Después de la segunda?

Otras funciones interesantes son las que se muestran:

```
Python 3.7.0a2 (default, Oct 18 2017, 18:58:26)
[GCC 7.2.0] on linux
Type "copyright", "credits" or "license()" for more information.
>>> round(3.5) #Redondea el número
4
>>> round(3.5555,2) #Redondea el número con dos decimales
3.56
>>> pow(3,2) #Potencia
9
>>> abs(-3) #Valor absoluto
3
```

Existen también funciones matemáticas para realizar cálculos más complejos, están concentradas en el módulo `math`. Para su utilización es necesario realizar una importación al módulo, se realiza mediante la instrucción[2]:

```
from math import *
```

En la página `https://docs.python.org/3/library/math.html` podemos analizar qué funciones hay disponibles y algunos ejemplos de uso. Algunas de ellas son[3]:

[2] También se puede usar **import modulo**, pero en ese caso para usar las funciones, debemos anteponer a la función el nombre del "modulo". Si lo importamos de esta forma, podemos acceder a las funciones que contiene con `help(math)`

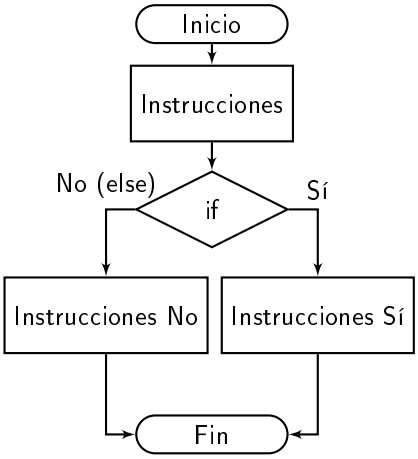
[3] Tal cual lo hemos importado podemos acceder la ayuda de una de ellas con, por ejemplo, `help(pow)`

Función	Ejemplo	Salida
Valor absoluto	fabs(-3)	3.0
Función exponencial	exp(1)	2.718281828459045
Potencial	pow(3,2)	9.0
Raíz cuadrada	sqrt(2.0)	1.4142135623730951
Coseno de un ángulo en radianes	cos(0)	1.0
Seno de un ángulo en radianes	sin(0)	0.0
Tangente de un ángulo en radianes	tan(0)	0.0
Conversión de radianes a grados	degree(pi)	180.0
Valor de pi	pi	3.141592653589793
Valor de e	e	2.718281828459045
Redondea un número al entero anterior	floor(3.4)	3
Redondea un número al entero posterior	ceil(3.4)	4

1.2.5. Control del flujo en Python: sentencia if .. else

Es la sentencia mas común para tomar una decisión:
if condición:
    <líneas de sentencias>

También podemos usar:
if condición_1:
    <líneas de sentencias>
elif condición_2:
    <líneas de sentencias>
elif condición_3:
    <líneas de sentencias>
...
else:
    <líneas de sentencias>



Ejemplos:

```
edad=18
if edad > 18:
    print("Adulto")

edad=20
if edad >= 20:
  print("Adulto")
  print("Qué mayor")

edad=18
if edad > 18:
    print("Adulto")
    print("Qué mayor")
else:
    print("Menor")
```

¿Qué obtendríamos con el siguiente código si introducimos 12? ¿y si introducimos -1?

```
edad=int(input("Introduce la edad: "))
if edad < 2 and edad > 0:
    print("Bebe")
elif edad < 12:
      print("Niño")
elif edad < 18:
      print("Joven")
else:
    print("Adulto")
```

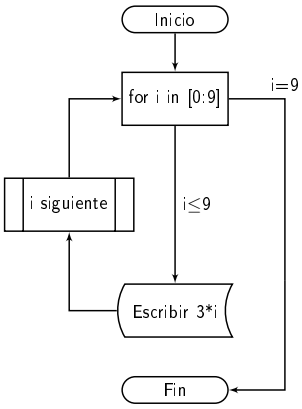
1.2.6. Bucles o iteraciones en Python

for Se suele usar cuando se necesita contar o realizar una acción un número determinado de veces.

for variable **in** elemento iterable (lista, cadena, range, etc.):
    <líneas de sentencias>

Ejemplos

```
for i in [0,1,2,3,4,5,6,7,8,9]:
    print(3*i)

for i in ("0123456789"):
    print(3*i)

numero = 25
for n in range(numero):
    print(numero-n,  end=", ")

print("\n")
for n in range(10,-10, -2):
    print(n)
```

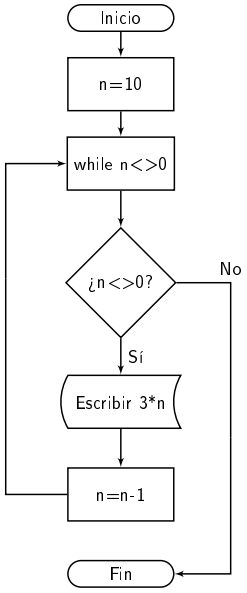


while

while condición:
    <líneas de sentencias>

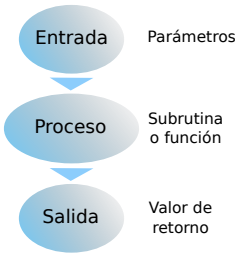
Ejemplos

```
n = 10
while n!=0:
    print(3, "*", n, " = ", 3*n)
    n = n-1

queda=36
while (queda % 2 == 0):
    queda=queda // 2
    print(queda)
```



1.2.7. Funciones

En computación, una subrutina o subprograma (también llamada procedimiento, función o rutina), como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.
Una subrutina al ser llamada dentro de un programa hace que el código principal se detenga y se dirija a ejecutar el código de la subrutina.
Las declaraciones de procedimientos generalmente son especificados por:



- Un nombre único en el ámbito: nombre de la función con el que se identifica y se distingue de otras. No podrá haber otra función ni procedimiento con ese nombre (salvo sobrecarga o polimorfismo en programación orientada a objetos).

- Si se trata de una función, un tipo de dato de retorno: tipo de dato del valor que la subrutina devolverá al terminar su ejecución.

- Una lista de parámetros: especificación del conjunto de argumentos (pueden ser cero, uno o más) que la función debe recibir para realizar su tarea.
- El código u órdenes de procesamiento: conjunto de órdenes y sentencias que debe ejecutar la subrutina. (`http://es.wikipedia.org/wiki/Subrutina`)

Las variables que se declaren en una función solo se usarán dentro de ellas. Cuando terminen se destruirán. Esto permite utilizar el mismo nombre de variable dentro de distintas funciones y su valor nunca se confundirá o mezclará. Para usar una variable declarada en una función fuera de su ámbito la debemos declarar global (**global**).

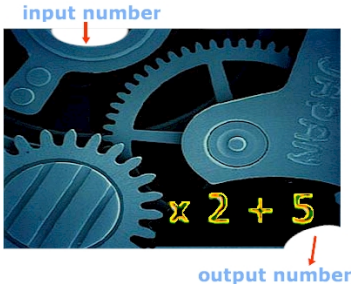
Ejemplos:
¿Qué diferencia hay entre ambas?

```python
def imprimir_mayor(x1, x2):
  if x1 <= x2:
    print(x2)
  else:
    print(x1)

n1=int(input("n1= "))
n2=int(input("n4= "))
imprimir_mayor(n1, n2)
```

```python
def imprimir_mayor(x1, x2):
  if x1 <= x2:
    return x2
  else:
    return x1

n1=int(input("n1= "))
n2=int(input("n4= "))
print(imprimir_mayor(n1, n2))
```

Para pensar: Escribe cómo sería una función que recibe dos números de argumento y devuelve en pantalla la media aritmética de ambos.

Ejemplos dos funciones que hacen lo mismo. ¿Las puedes explicar?




```python
#Cálculo del factorial de un número (iterativo)
def factorial_iterativa(numero):
    factorial=1
    for contador in range(numero,1, -1):
        factorial = factorial * contador
    return factorial

#Cálculo del factorial de un número (recursiva)
def factorial_recursiva(numero):
    if numero == 0:
        return 1 #El factorial de 0 es 1
    elif numero > 0:
        return numero * factorial_recursiva(numero - 1)
```

Para pensar cómo podría ser una función a la que se le pasen dos números naturales (base y exponente) y devuelva como resultado la potencia

$$base^{exponente}$$

2. Actividades

2.1. Criba de Eratóstenes en modo texto

Diseña una solución para consola que simule la criba de Eratóstenes para buscar los números enteros y positivos primos que hay hasta un número entero positivo dado de antemano.

Vamos a programar un método propuesto por Eratóstenes[4] para obtener la lista de todos los primos menores, hasta cierto valor.

Los pasos de la función son:

1. Primero hacemos una lista de todos los números.
2. Tachamos el 0 y 1.
3. Tomamos el 2 y tachamos todos los múltiplos de 2 en la lista.

   Para cada primo p, el múltiplo (propio) más pequeño que no está tachado es p^2. Así que los primos menores que la raíz cuadrada de n pueden tener múltiplos sin tachar en la tabla, pero los mayores no. ¿Es cierto esto? ¿Por qué?.
4. Buscamos el primer número sin tachar y tachamos todos sus múltiplos mayores.
5. Repetimos el paso anterior hasta el siguiente entero de la raíz cuadrada del valor.
6. Los que quedaron sin tachar son los que no tienen divisores (propios) o sea los primos.

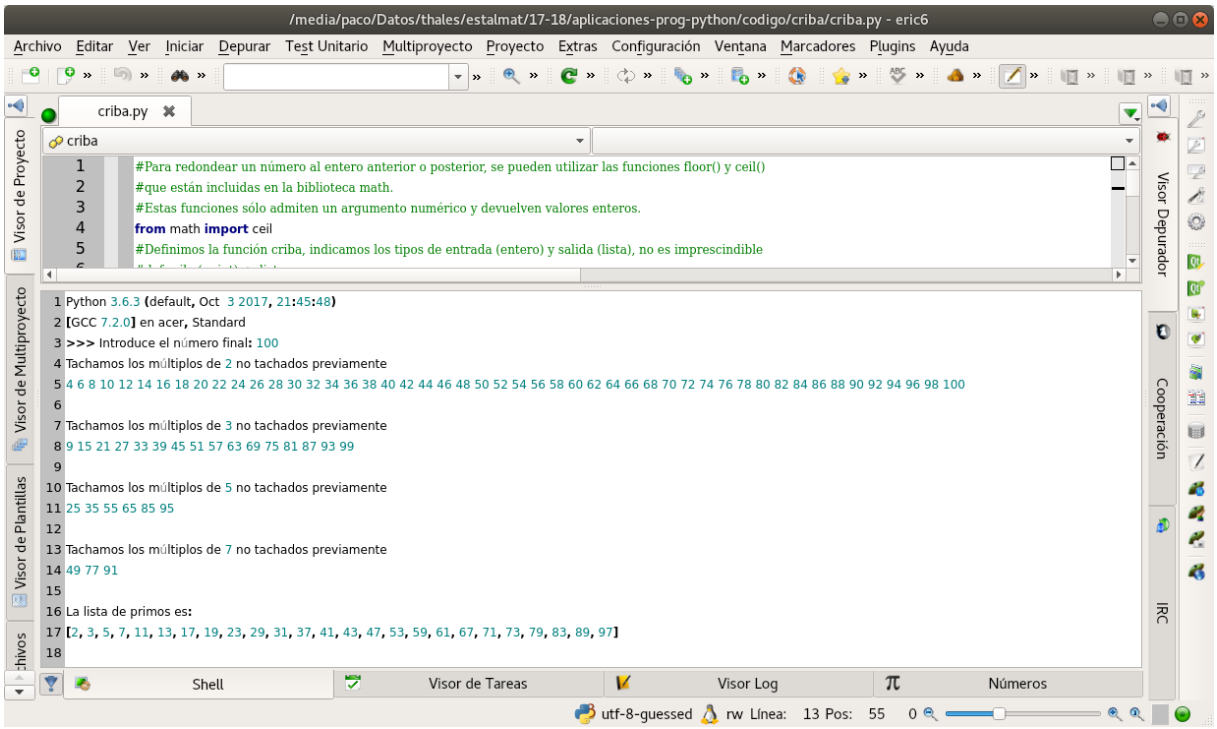
Solución Iniciamos eric6 y la primera vez nos pide algunas cuestiones de configuración.

```
1  u"""
2  Para redondear un número al entero anterior o posterior, se pueden utilizar
3  las funciones floor() y ceil() que están incluidas en la biblioteca math.
4  Estas funciones solo admiten un argumento numérico y devuelven valores enteros.
5  Como vamos a usar ceil(), la importamos del módulo.
6  """
7  from math import ceil
8
9  # Definimos la función criba, indicamos el tipo de entrada (entero),
10 # no es imprescindible.
11
12 def criba(n: int):
13     u"""Criba de Eratatóstenes."""
14     # Construimos una lista de tamaño el número final pasado a la función con False
       ↪  o True
15     # Ya sabemos que el 0 y el 1 no son primos los ponemos a False, el resto a True
16     # Como las listas comienzan con el índice 0, para almacenar n números llegamos
       ↪  hasta n-1
17     primos = 2*[False] + (n-1)*[True]
18     # Para hacer las comprobaciones, comenzamos en el 2
19     # Solo llegamos hasta el siguente entero de la raíz cuadradra del nº
20     for i in range(2, int(ceil(n**0.5))):
```

[4] Eratóstenes (Cirene, 276 a. C. - Alejandría, 194 a. C.) fue un matemático, astrónomo y geógrafo griego. Uno de sus resultados más importantes es que determinó con un error más que aceptable el radio de la Tierra (¡ya sabían que era redonda!). Eratóstenes hizo un mapa del mundo conocido. Considerando la limitación de los medios de transporte de la época, su mapa del área mediterránea es muy apreciable.

```
21          if primos[i*i] is True:
22              print("Tachamos los múltiplos de",  i,  "no tachados previamente")
23              # Comenzamos a tachar desde el cuadrado del número
24              for j in range(i*i, n+1, i):
25                  # Solo lo tachamos si no se han tachado ya
26                  if primos[j] is True:
27                      primos[j] = False
28                      # Imprimimos el índice del nº tachado. De separación usamos un
                            ↪  espacio
29                      print(j,  end=' ')
30              # Realizamos un salto de línea
31              print("\n")
32      print("La lista de primos es:")
33      # Construimos una lista con el elemento primo
34      # Iteramos sobre una secuencia con el for
35      # Usando la función enumerate(), obtenemos el índice de posición (primo)
36      # junto a su valor (valor) correspondiente
37      #Solo seleccionamos los índices de posición de la lista primos con valor True
38
39      print([primo for primo, valor in enumerate(primos) if valor])
40
41  if __name__ == '__main__':
42      # Almacenamos en la variable final el nº final de la Criba
43      final = int(input("Introduce el número final: "))
44      # Ejecutamos la función
45      criba(final)
```

Escribimos el código[5] anterior en eric6, guardamos el fichero desde Archivo > Guardar o con ctrl + S y ejecutamos el programa con F2 o desde Iniciar > Ejecutar Script



En la captura anterior hemos probado el programa con n=100

[5]Para entender qué significa la última parte del código véase `http://docs.python.org.ar/tutorial/3/modules.html`

2.2. El módulo turtle

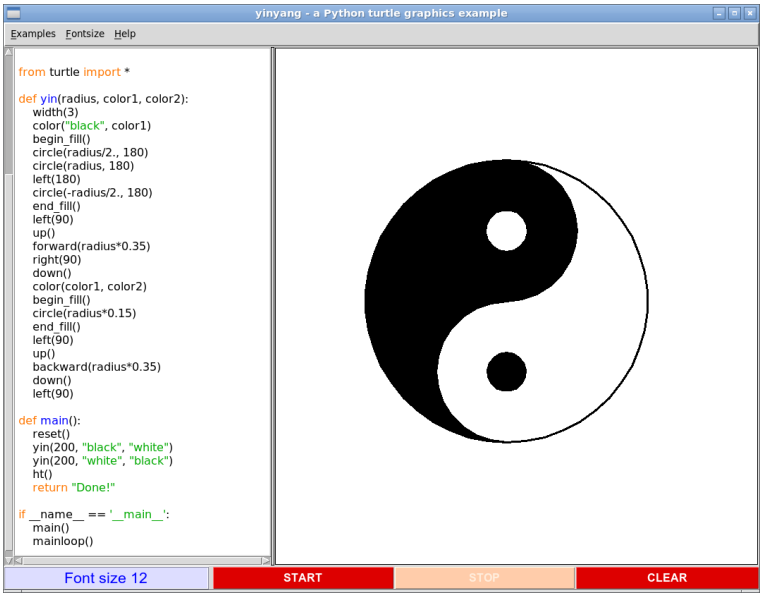
Python permite con una serie de librerías la implementación de interfaces gráficas de usuario. Antes de comenzar con una de ellas vamos a crear aplicaciones gráficas muy sencillas con una «tortuga».

Podemos acceder a la documentación del módulo:

- `https://docs.python.org/3.7/library/turtle.html` o de la bibliografía [2, 6]
- Si queréis ver algunos ejemplos[6] de lo que se puede hacer solo hay que ejecutar desde un terminal:

```
$ python3 -m turtledemo
```

Por ejemplo:



Para utilizar el módulo turtle solo hace falta importarlo. Si se va a escribir código no orientado a objetos:

```
from turtle import *
```

Si se va a escribir código orientado a objetos:

```
import turtle
```

Antes de comenzar os mostramos algunos de los métodos básicos de dibujo con la tortuga:

forward(x) avanza *x* pasos[7].

backward(x) retrocede *x* pasos.

left(x) gira a la izquierda *x* grados.

right(x) gira a la derecha *x* grados.

penup() levanta el lápiz, al desplazar el lápiz ya no se dibujan segmentos.

pendown() baja el lápiz, al desplazar el lápiz vuelve a dibujar.

showturtle() muestra el cursor (la tortuga).

hideturtle() oculta el cursor (la tortuga).

shape("turtle") establece la forma de nuestra tortuga, en este caso, resulta tener la forma de una tortuga real en lugar de una flecha. Podemos usar: “arrow”, “turtle”, “circle”, “square”, “triangle”, “classic”.

[6] Hay que tener instalado los paquetes `python3.x-examples` e `idle-python3.x`

[7] Los métodos forward, backward, left y right permiten controlar a la tortuga con coordenadas y ángulos relativos a la posición y orientación de la tortuga. Si decimos dos veces forward(100), la tortuga habrá avanzado 200 pasos desde la posición de partida en la dirección a la que apunta.

Las posiciones en el área de dibujo se localizan mediante coordenadas (x, y) en el que cada píxel es una unidad y en el que el origen en el centro de la ventana.

Estos otros métodos permiten controlar a la tortuga con valores absolutos:

goto(x,y) ubica a la tortuga en la posición de coordenadas (x, y). Si se escriben varias instrucciones goto(), se van dibujando los segmentos uno a continuación del otro.

setx(x) ubica a la tortuga en la posición de abscisa x y la misma ordenada actual.

sety(y) ubica a la tortuga en la posición de ordenada y y la misma abscisa actual.

setheading(x) hace que la tortuga apunte en dirección x grados.

towards(x, y) hace que la tortuga apunte en dirección al punto (x, y).

home() ubica a la tortuga en la posición de coordenadas $(0, 0)$.

Podemos controlar algunos elementos del aspecto de las líneas, como el grosor del trazo y el color:

pensize(x) usa un lápiz con trazo de x píxeles de grosor.

pencolor('color') usa el color *color* para los trazos (cadena con el nombre del color en inglés)

bgcolor('color') color de fondo de la zona de dibujo.

También disponemos de:

dot(x, 'color') dibuja un punto de diámetro x centrado en la posición actual con el color pasado como parámetro (es opcional).

circle(x) dibuja un círculo de radio x. El círculo está centrado a x pasos a la izquierda de la tortuga.

write("texto") escribe el texto en pantalla, en la posición actual de la tortuga.

mainloop() se escribe al final del programa, hace que el programa no termine, sino que permanezca a la espera de los eventos definidos en el programa. Permite crear programas de tortuga interactivos.

begin_fill() indica a Python que las figuras que se dibujen a partir de ese momento se deben rellenar.

end_fill() indica a Python que las figuras deben dejar de rellenarse.

fillcollor(color) permite establecer el color de relleno.

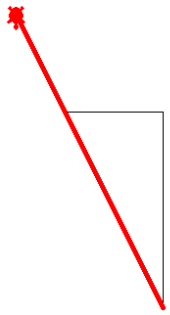
- Para realizar lo que sigue podemos usar el modo interactivo o la shell integrada que nos facilita eric6.

2.2.1. En modo interactivo

En nuestro caso, como estamos comenzando usaremos:

```
1  Python 3.6.7 (default, Oct 22 2018, 11:32:17)
2  [GCC 8.2.0] en acer, Standard
3  >>> import turtle # Python está listo para recibir órdenes de turtle.
4  >>> help() # Aparece el prompt de la ayuda
5  help>
6  help> turtle #Sale la ayuda del módulo turtle
7  ...
8  help> for # Ayuda de for
9  ...
10 help> q # Salimos de la ayuda
11 >>>
```

```
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] en acer, Standard
>>> import turtle # Python está listo para recibir órdenes de turtle.
>>> turtle.forward(100) # Se abre una nueva ventana, en ella la tortuga ha dibujado
 ↪  un segmento de longitud 100 pixeles.
>>> turtle.shape('turtle') # Cambiamos la forma de nuestra tortuga.
>>> turtle.right(90) # La tortuga giró hacia la derecha 90 grados.
>>> turtle.forward(200) # Dibujamos un segmento de longitud 200 pixeles.
>>> turtle.color('red') # Cambiamos el color a rojo.
>>> turtle.width(5) # Cambiamos el grosor a 5 pixel.
>>> turtle.goto(-50,100) # Nos desplazamos a la posición (-50,100).
```



2.2.2. Usando eric6 u otro editor

Creamos un nuevo fichero con eric6 que guardaremos con el nombre `tortuga2.py3`. En la pantalla blanca escribimos el programa:

```
from turtle import *
forward(100)
mainloop()
```

Una vez escrito y guardado lo ejecutamos con la tecla F2, el resultado será:

Un cuadrado

```
# Un cuadrado parametrizado
from turtle import *
a = 100
i = 1
while i <= 4:
    forward(a)
    right(90)
    i = i+1
mainloop()
```

Una vez escrito y guardado lo ejecutamos con la tecla F2

Una estrella

```
import turtle
for i in range(5):
    turtle.forward(100)
    turtle.right(144)
turtle.mainloop()
```

Más programas

```
from turtle import *
color('red', 'yellow')

begin_fill()
while True:
    forward(200)
    left(170)
    # Termina si el módulo del vector de posición es menor de 1
    # Es decir, cuando volvemos (de forma aproximada) a la posición inicial (0,0)
    if abs(pos()) < 1:
        break
end_fill()
mainloop()
```

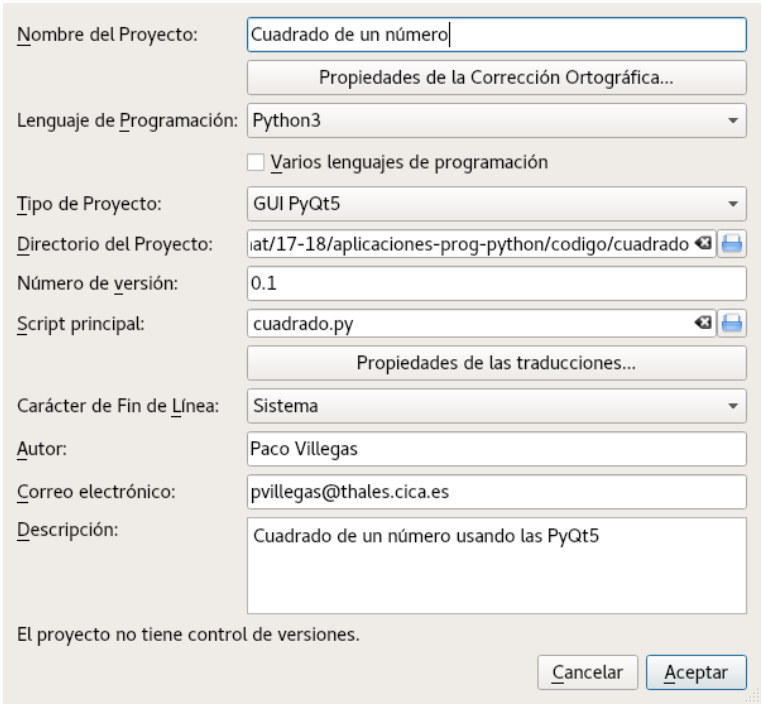
```
import turtle
colores = ['red', 'purple', 'blue', 'green', 'yellow', 'orange' ]
turtle.bgcolor('black')
for x in range(360):
    turtle.pencolor(colores[x % 6])
    turtle.width(x/100+1)
    turtle.forward(x)
    turtle.left(59)
turtle.mainloop()
```

```
from turtle import  *
pencolor('red')
pensize(4)
shape('turtle')
for  n in range(10) :
    penup()
    goto(0,-n*20)
    pendown()
    circle(20*n)
mainloop()
```

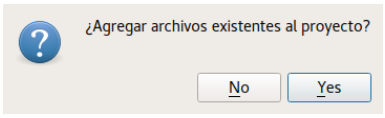
¿Puedes explicarlos?

2.3. Nuestro primer programa gráfico con QT Designer: cuadrado de un número

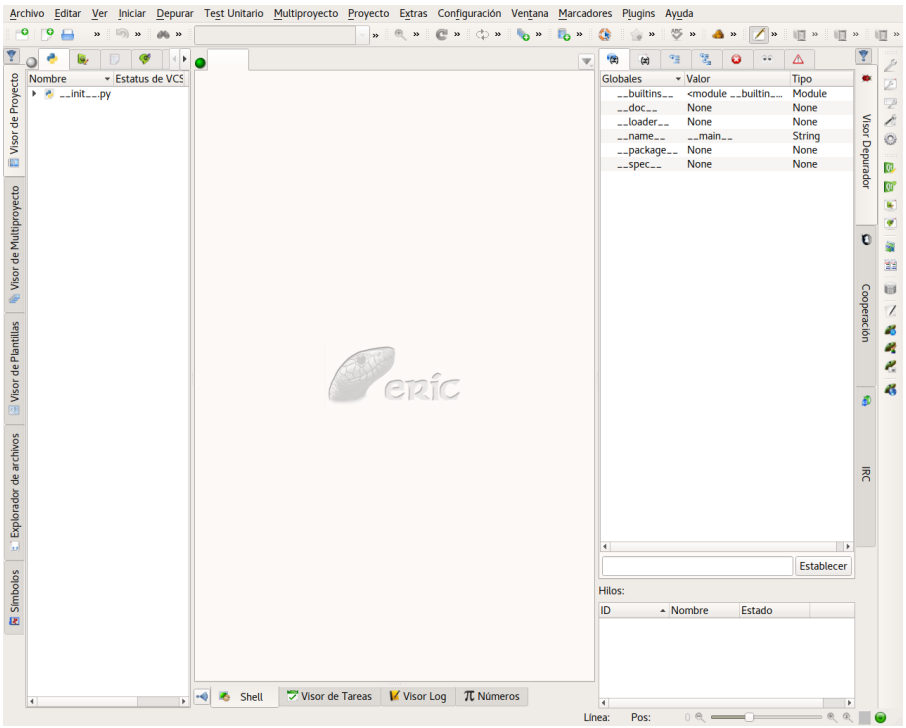
Creamos el proyecto desde Proyecto » Nuevo (véase el gráfico **1.1 en la página 4**), hay que seleccionar/crear la carpeta en la que trabajar:



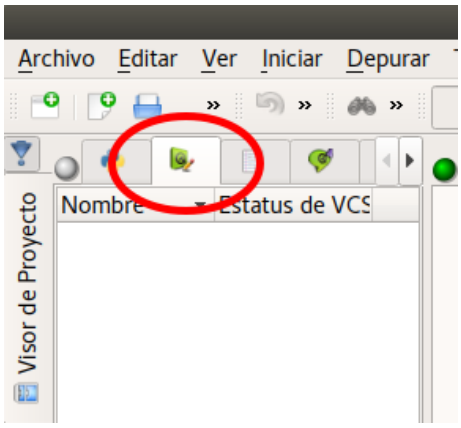
Seleccionamos que no



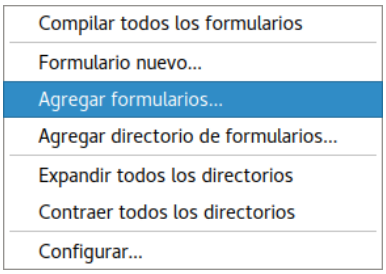
Tenemos



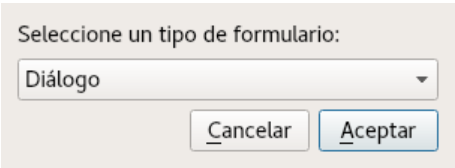
Pulsamos sobre **Formularios**



en la parte de abajo (parte blanca) pulsamos con el botón derecho del ratón y se nos mostrará la ventana emergente:

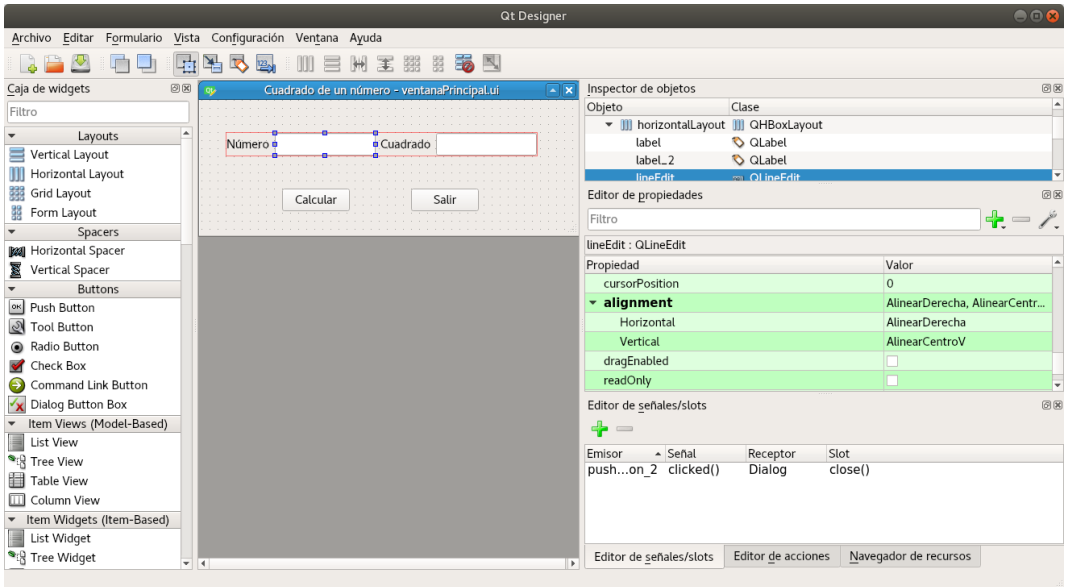


Optamos por **formulario nuevo**, en él seleccionamos directamente que sea el tipo Dialogo (Widget), nos aparece por defecto.



Después, creamos la carpeta «ui» y en ella el formulario de nombre **ventanaPrincipal**, el programa le añade la extensión de forma automática.

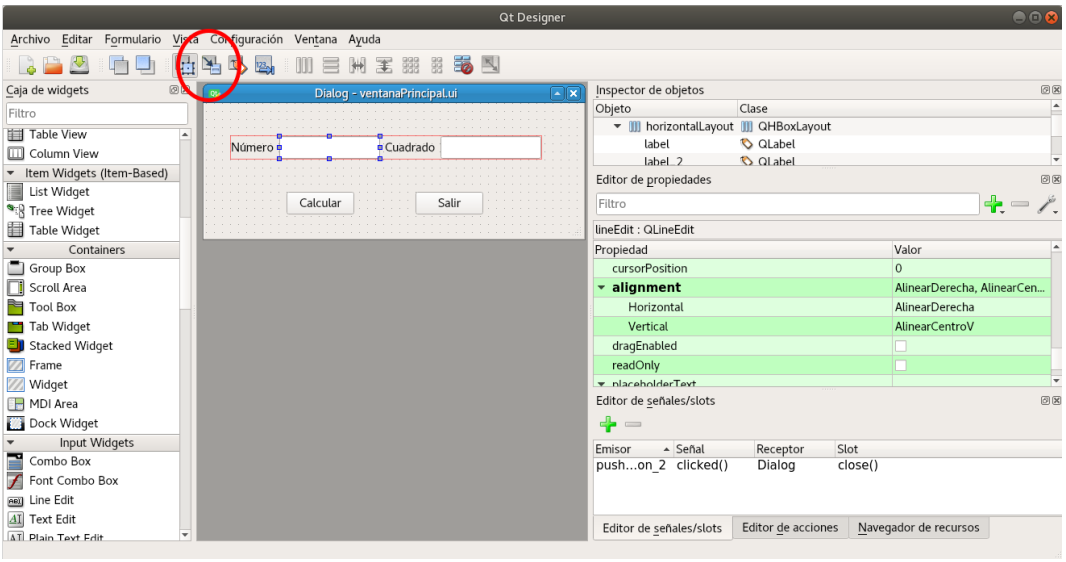
Una vez creado añadimos dos campos de la Caja de Widgets (**LineEdit**), uno para introducir el número y otro para obtener su cuadrado.



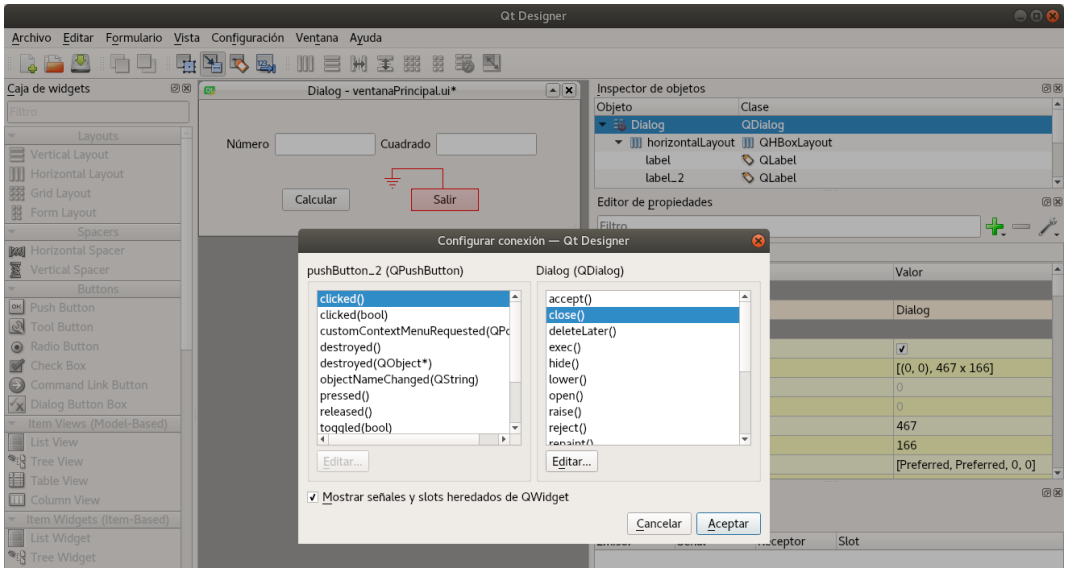
Cambiamos los valores por defecto para que la alineación sea a la derecha (**alignment**) y que el segundo sea de solo lectura (**readOnly**).

Solo ajustamos un poco el tamaño, la posición de los botones (**PushButton**) y añadimos un par de etiquetas (**Label**).

Para conseguir que el botón **Salir** realice el efecto deseado, pulsamos sobre **Editar señales/slot** (el icono está marcado en la captura que sigue)



Después, pulsamos sobre el botón **Salir** y arrastramos hasta el Widget[8] principal (se mostrará en rojo), y tras seleccionar Mostrar señales y slots heredados de QWidget, haremos que al pulsar sobre el botón (evento `clicked()`), se cierre la ventana principal (`close()`)[9]



[8] Widget es una contracción de windows gadgets, es decir una contracción de “artilugios de ventanas”.

[9] En general, todas las aplicaciones gráficas que creemos ejecutan sus acciones mediante eventos. Los eventos pueden ser iniciados por el usuario (click en un botón, se pulsa un tecla), por el sistema (se introduce un CD), etc. En consecuencia, además de crear los elementos que se necesitan en la aplicación (botones, etiquetas, ...) debemos programar el código que responda a los eventos. En PyQt, lo usual es manejar los eventos mediante Signals (señales) y Slots.

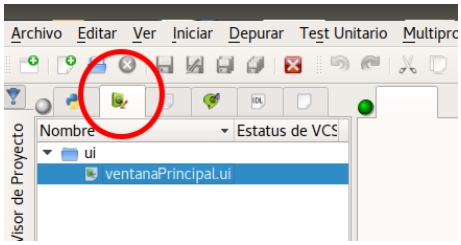
- Una señal es un mensaje.
- Un slot es un receptor de esos mensajes.

Cuando se pulsa el botón “Salir”, el objeto QPushButton correspondiente emite la señal clicked() que conectamos con el objeto QDialog (sería el slot) para que al recibir la señal se cierre (close()) la ventana principal. Si no conectamos la señal con el objeto receptor, no pasaría nada, las señales por sí solas no tienen efectos.

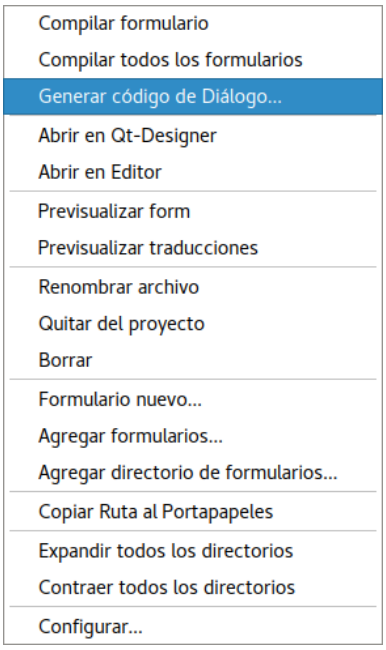
Por último, cambiamos el título de la ventana principal por "Cuadrado de un Número" (propiedad **windowTitle**).

Guardamos en la carpeta de nuestro proyecto (desde Archivo > Guardar). Salimos de Qt Dessigner y volvemos a eric6.

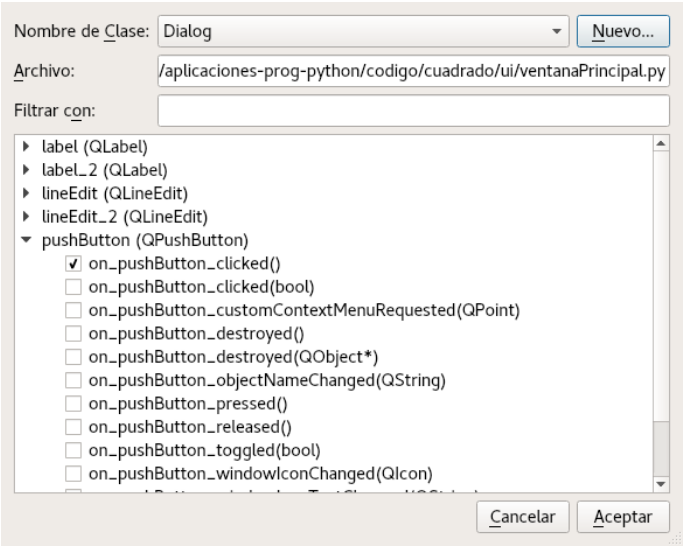
Seguimos en Formularios



Lo seleccionamos y pulsando con el botón derecho del ratón sobre él se nos abre el menú:

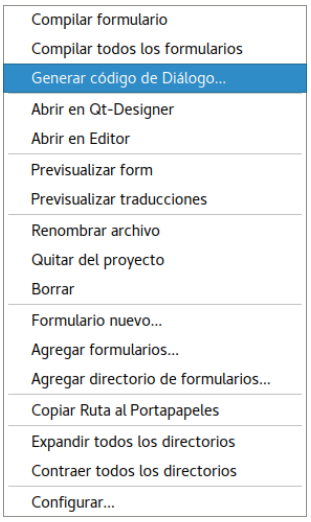


Vamos a crear el "Código de diálogo" del botón que nos calculará el cuadrado (evento `clikked()`)

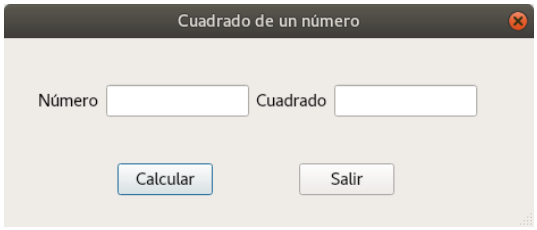


En el nombre de la clase, pulsamos sobre **Nuevo** y aceptamos los valores que nos aparecen (`Dialog`). Marcamos como se ve en la captura anterior.

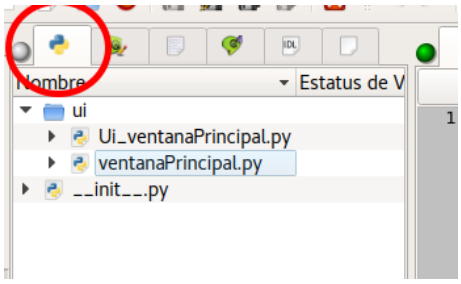
Después, compilamos el Formulario (primera línea del menú de la captura). Cada vez que lo cambiemos debemos ejecutar esta acción (compilar el formulario).



Desde aquí, por ejemplo podemos previsualizar cómo quedaría.



Nos volvemos a «Fuentes»



Si hacemos doble click sobre cualquiera de ellos podremos acceder al código que tenemos generado.

Es hora de añadir el código de Python para obtener el cuadrado del nº. En el fichero **ventanaPrincipal.py** añadimos/modificamos el código:

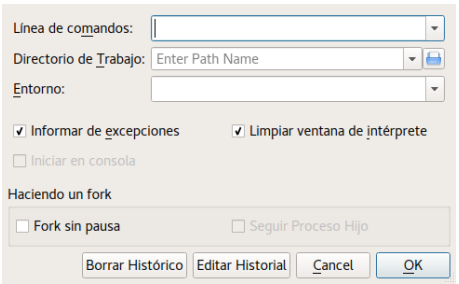
```
1  # -*- coding: utf-8 -*-
2
3  """Module implementing Dialog."""
4
5  from PyQt5.QtCore import pyqtSlot
6  from PyQt5.QtWidgets import QDialog
7  #Añadimos
8  from PyQt5 import QtGui
9  from .Ui_ventanaPrincipal import Ui_Dialog
10
11 class Dialog(QDialog, Ui_Dialog):
12
```

```
    """Class documentation goes here."""

    def __init__(self, parent=None):
        """
        Constructor.

        @param parent reference to the parent widget
        @type QWidget

        """
        super(Dialog, self).__init__(parent)
        self.setupUi(self)
        #Para que valide que solo introducimos números reales
        self.lineEdit.setValidator(QtGui.QDoubleValidator())

    @pyqtSlot()
    def on_pushButton_clicked(self):
        """Slot documentation goes here."""
        # TODO: not implemented yet
        #Comentamos la línea que sigue
        #raise NotImplementedError
        numero = float(self.lineEdit.text())
        numero = numero*numero
        self.lineEdit_2.setText(str(numero))
```

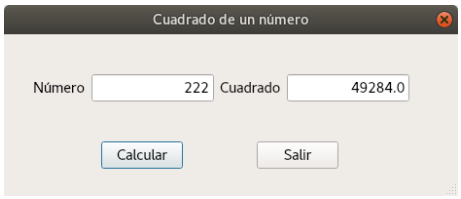
Editamos el fichero de nombre **cuadrado.py** y añadimos en el fichero (estará en el raíz de nuestro proyecto) el código:

```
from PyQt5 import QtWidgets

from ui.ventanaPrincipal import Dialog

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    ui = Dialog()
    ui.show()
    sys.exit(app.exec_())
```

Si ahora pulsamos sobre ⇧ + F2 ya podemos ejecutarlo (o desde Iniciar - Ejecutar Proyecto o desde los accesos rápidos), veremos primero una ventana para pasarle parámetros



y si optamos por los valores por defecto:



Antes de seguir, comentaremos un poco qué significado tienen las líneas del script principal **cuadrado.py**. Estas líneas se repetirán casi de forma literal en los diferentes programas.

1 Importa desde `PyQt5` el módulo `QtWidgets`. En él tenemos las clases que proporcionan un conjunto de elementos para crear interfaces de usuario clásicas de escritorio.

3 Desde el módulo `ui.ventanaPrincipal` importa la clase `Dialog`.

5 Básicamente, usando

```
if __name__ == "__main__"
```

lo que hacemos es ver si el módulo ha sido importado o no. Si no se ha importado (se ha ejecutado como programa principal) ejecuta el código dentro del condicional. No es imprescindible y podemos no usarlo.

6 Importa los parámetros del sistema, son necesarios para iniciar la aplicación.

7 Creamos nuestro objeto aplicación, que llamamos `app` a partir de la clase `QApplication` del módulo `QtWidgets`. Le pasamos (mediante `sys.argv`) los argumentos de la línea de comandos usados al ejecutar el programa para que pueda inicializar de forma adecuada la aplicación.

8 Creamos un objeto llamado `ui` a partir de la clase Dialog() importada en la línea 3.

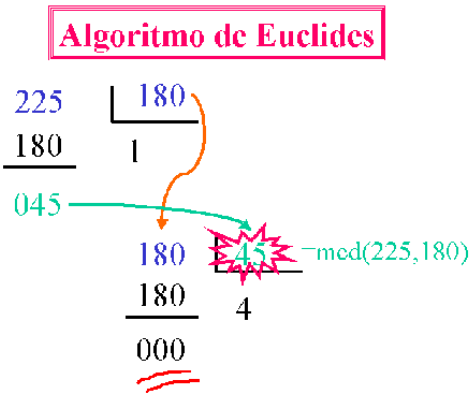
9 Visualizamos el objeto `ui` mediante el método `show()`.

10 Con el método `exec_()` conseguimos que nuestra aplicación (`app`) entre en un bucle a la espera de que se ejecuten acciones. `sys.exit()` permite salir de aplicación de la forma correcta (libera los recursos usados cuando se cierra la ventana). `sys.exit()` no es imprescindible, aunque sí recomendable usarlo.

☺ Realiza un programa similar al anterior que calcule el factorial de un número pasado como argumento.

2.4. Cálculo del máximo común divisor y el mínimo común múltiplo mediante el Algoritmo de Euclides

"El **algoritmo de Euclides** es un método antiguo y eficaz para calcular el máximo común divisor (**MCD**). Fue originalmente descrito por Euclides en su obra *Elementos*. ... Este algoritmo tiene aplicaciones en diversas áreas como álgebra, teoría de números y ciencias de la computación entre otras. Con unas ligeras modificaciones suele ser utilizado en computadoras electrónicas debido a su gran eficiencia. (`http://es.wikipedia.org/wiki/Algoritmo_de_Euclides`)"



Calculemos el máximo común divisor de 105 y 12

Paso	División Euclídea	Significado
1	$105 = 12 \cdot 8 + 9$	mcd(105,12)=mcd(12,9)
2	$12 = 9 \cdot 1 + 3$	mcd(12,9)=mcd(9,3)
3	$9 = 3 \cdot 3 + 0$	mcd(9,3)=mcd(3,0)=3

$$a \cdot b = MCD(a,b) \cdot MCM(a,b)$$

Analicemos en primer lugar el código de las dos funciones que vamos a utilizar:

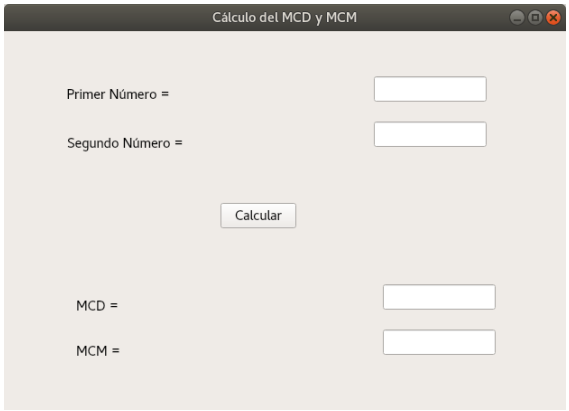
```
1  def mcd(a,b):
2      if b == 0:
3          return a
4      else:
5          return mcd(b, a%b)
6
7  def mcm(a, b):
8      mcd_ab=mcd(a, b)
9      if mcd_ab==0:
10         return 0
11     else:
12         return (a*b)//mcd_ab
```

☺ ¿Funcionará una llamada del tipo mcd(7,3)? ¿y del tipo mcd(3,7)?

Es el momento de comenzar nuestro programa:

1. Creamos nuestro Formulario como se ve en el gráfico de al lado. Solo usaremos los widgets analizamos en el ejercicio anterior.

2. Crearemos cuatro **LineEdit**, dos para introducir los números y dos para obtener los cálculos.
   - Las cuatro con alineación derecha (Right).
   - Las dos últimas de tipo "Solo lectura".



3. Insertamos cuatro etiquetas de texto (**Label**) para mostrar:
   a) Primer número =
   b) Segundo número =
   c) MCD =
   d) MCM =

   El nombre que demos a las etiquetas no es importante para este programa.

4. Añadimos un botón (**PushButton**) con las características del gráfico anterior.

5. Completamos el código tal cual sigue:

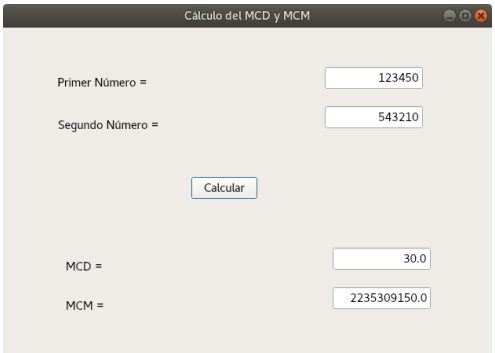
```
"""
Module implementing Dialog.
"""

from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import QDialog

from .Ui_ventanaPrincipal import Ui_Dialog
#Añadimos para poder validar los datos de entrada
from PyQt5 import QtGui

class Dialog(QDialog, Ui_Dialog):
    """
    Class documentation goes here.
    """
    def __init__(self, parent=None):
        """
        Constructor

        @param parent reference to the parent widget
        @type QWidget
        """
        super(Dialog, self).__init__(parent)
        self.setupUi(self)
        #para validar que solo introducimos números enteros
        self.lineEdit.setValidator(QtGui.QIntValidator())
        self.lineEdit_2.setValidator(QtGui.QIntValidator())

    @pyqtSlot()
    def on_pushButton_clicked(self):
        """
        Slot documentation goes here.
        """
        # TODO: not implemented yet
        #comentamos raise NotImplementedError
        a=int(self.lineEdit.text())
        b=int(self.lineEdit_2.text())
        self.lineEdit_3.setText(str(mcd(a, b)))
        self.lineEdit_4.setText(str(mcm(a, b)))
```

✔ Que no se nos olvide, **debajo del código anterior tendremos que tener el código de las funciones que nos permiten calcular el MCD y MCM.**

✔ Tendremos que adecuar el fichero **mcd.py** de la misma forma que lo hicimos en el ejercicio anterior.

Para conseguir que solo admita números enteros en la entrada de las casillas usamos:

```
self.lineEdit.setValidator(QtGui.QIntValidator())
self.lineEdit_2.setValidator(QtGui.QIntValidator())
```



2.5. Criba de Eratóstenes en modo gráfico.

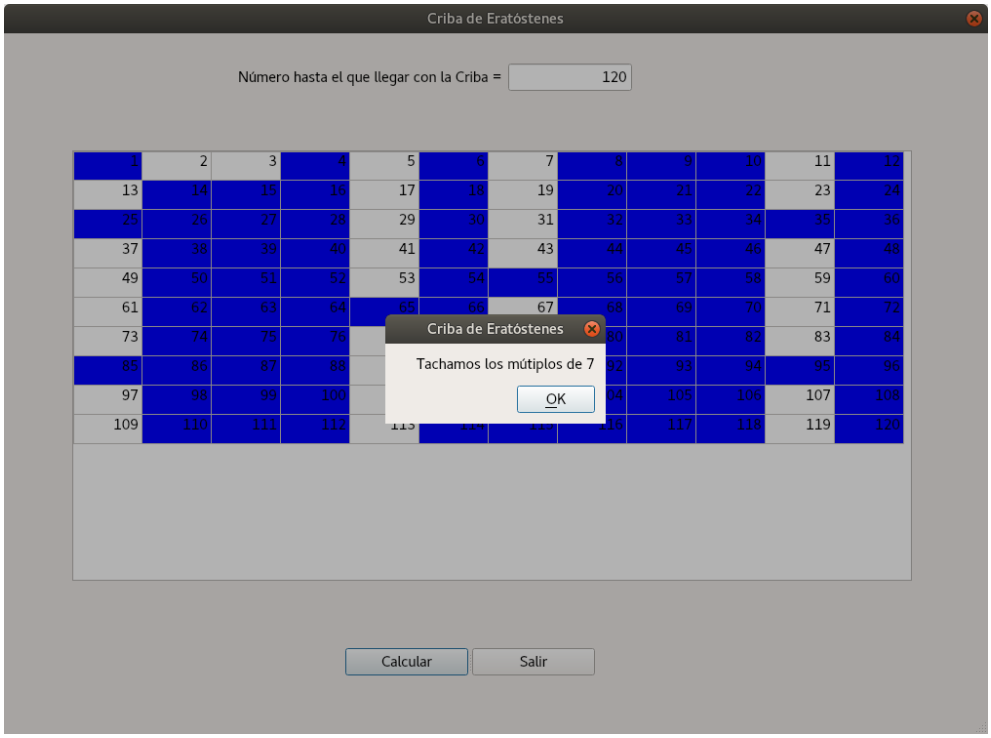
Diseña una solución gráfica, como el gráfico de abajo, para simular la criba de Eratóstenes.

Solución

Diseña un formulario con

- una etiqueta.
- un cuadro de texto con alineación derecha.
- un botón para que inicie el proceso.
- otro botón para Salir.
- un **TableWidget** para mostrar los números.
  - En sus propiedades marcamos que no se muestren ni los encabezados de las filas ni las columnas (desmarcar **horizontalHeaderVisible** y **verticalHeaderVisible**)
  - También que no permita la edición de las celdas (**editTriggers** en **NoEditTriggers**).

	2	3	~~4~~	5
~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~
~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~



Código:

```
1 # -*- coding: utf-8 -*-
2
3 """
4 Module implementing Dialog.
5 """
6 #Añadimos algunos módulos que no se importan por defecto
```

```
from PyQt5 import QtCore,  QtGui
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import QDialog,  QMessageBox
from PyQt5.QtWidgets import QTableWidgetItem
from math import ceil

from .Ui_ventanaPrincipal import Ui_Dialog

class Dialog(QDialog, Ui_Dialog):
    """
    Class documentation goes here.
    """
    def __init__(self, parent=None):
        """
        Constructor

        @param parent reference to the parent widget
        @type QWidget
        """
        super(Dialog, self).__init__(parent)
        self.setupUi(self)
         #para validar que solo introducimos números enteros
        self.lineEdit.setValidator(QtGui.QIntValidator())

    @pyqtSlot()
    def on_pushButton_clicked(self):
        """
        Slot documentation goes here.
        """
        # TODO: not implemented yet
        #raise NotImplementedError
        #Almacenamos en la variable numero el valor del LineEdit
        numero=int(self.lineEdit.text())
        # Guardamos en la variable el ancho en pixel del tableWidget
        anchoTabla = self.tableWidget.frameGeometry().width()
        # Así podemos obtener el altoo
        # alto = self.tableWidget.frameGeometry().height()
        # Columnas para mostrar los números
        columnas=12
        # Ancho que daremos a las columnas a partir del ancho del QTableWidget y del
        ↪  nº de columnas
        anchoColumnas=anchoTabla/columnas
        # Filas necesarias si establecemos el nº de columnas a 12
        if  numero/columnas==numero//columnas:
            filas=numero//columnas
        else:
            filas=numero//columnas+1
        # Creamos la tabla con el nº de filas adecuado.
        self.tableWidget.setRowCount(filas)
        self.tableWidget.setColumnCount(columnas)
        # Variable para ir guardando los números que pondremos en la tabla
        llenalista=1
```

```
        # Escribimos los números en la tabla
        for i in range(filas):
            for j in range(columnas):
                self.tableWidget.setItem(i,j, QTableWidgetItem(str(llenalista)))
                # Para alinear el texto a la derecha
                self.tableWidget.item(i,j).setTextAlignment(QtCore.Qt.AlignRight)
                #Para establecer el ancho  de las columnas tenemos dos opciones
                #Así lo ajustamos a los anchos de los valores introducidos
                #self.tableWidget.resizeColumnToContents(j)
                # Así establecemos el ancho en pixel
                self.tableWidget.setColumnWidth(j, anchoColumnas)
                llenalista=llenalista+1
                # Para completar solo hasta el número
                if llenalista>numero:
                    break
            if llenalista>numero:
                break
        # El 1 no es primo, así que lo tachamos (fondo azul)
        self.tableWidget.item(0,0). setBackground(QtGui.QColor('blue'))
        # Creamos un conjunto para ir almacenando los números tachados
        # es uno de los tipos de datos más rápidos para comprobar si
        # un elemento pertenece a él
        tachados=set()
        for i in range(2, int(ceil(numero**0.5))):
            if i not in tachados:
                # Creamos una ventana emergente. La hemos creado a "pelo"
                # Se pueden crear desde eric (Extras -> Asistentes)
                QMessageBox.about(self, "Criba de Eratóstenes", "Tachamos los
                ↪ mútiplos de "+str(i))
                for j in range(i*i, numero+1, i):
                    tachados.add(j)
                    if (j % columnas) != 0:
                        s=j // columnas
                        t=(j % columnas)-1
                    else:
                        s=j // columnas-1
                        t=columnas-1
                    self.tableWidget.item(s,t). setBackground(QtGui.QColor('blue'))
```

Tened en cuenta que el script principal del proyecto tiene que contener:

```
from PyQt5 import QtWidgets

from ui.ventanaPrincipal import Dialog

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    ui=Dialog()
    ui.show()
    sys.exit(app.exec_())
```

Índices de la tabla:

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	0,10	0,11
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11
...											

Valores almacenados: j

1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60
...											

¿Cómo calculamos la posición (s,t) a partir del número?

Divisor entero (//) y resto (%) del número entre 12											

j es el número

```
if j % columnas != 0:
        s = j // columnas                                  → Fila
        t = (j % columnas) – 1                             → Columna
else:
        s = j // 12 – 1                                    → Fila
        t = columnas-1                                     → Columna
self.tableWidget.item(s,t). setBackground(QtGui.QColor('blue'))   → Lo pintamos de azul
```

Figura 1: Ficha de apoyo

Ejercicio: ¿Eres capaz de adaptar los programas anteriores con las siguientes características?:

1. En vez de usar un QDialog usa un QMainWindow.
2. Modifica los nombres de los widgets usados con nombres más descriptivos. Por ejemplo en vez de push-

Button se podría usar pushButtonSalir, pushButtonCalcular o para lineEdit el nombre de lineEditPrimerNumero. Adecúa el código obtenido para esos nombres.

2.6. QGraphicsView: eventos, líneas y coordenadas

2.6.1. QGraphicsView

"QGraphicsView[10] es el framework dentro de Qt que permite la creación e interacción de elementos gráficos 2D y que usa el método de programación modelo/vista. De manera simple, varias vistas pueden observar una misma escena y una escena puede contener elementos (items) de diferentes formas geométricas.

El framework está compuesto de 3 elementos:

- QGraphicsScene (La Escena):

  Representa una escena con items. Es la clase que se encarga de almacenar los widgets, así como manejar-propagar eventos a cada item.

  Además esta clase se encarga de manejar los estados de un item. Un objeto QGraphicsScene es muy flexible como para incluir cualquier número de objetos QGraphicsItem.

- QGraphicsView (La Vista):

  La clase que se encarga de proporcionar los widgets que visualizan el contenido de una escena.

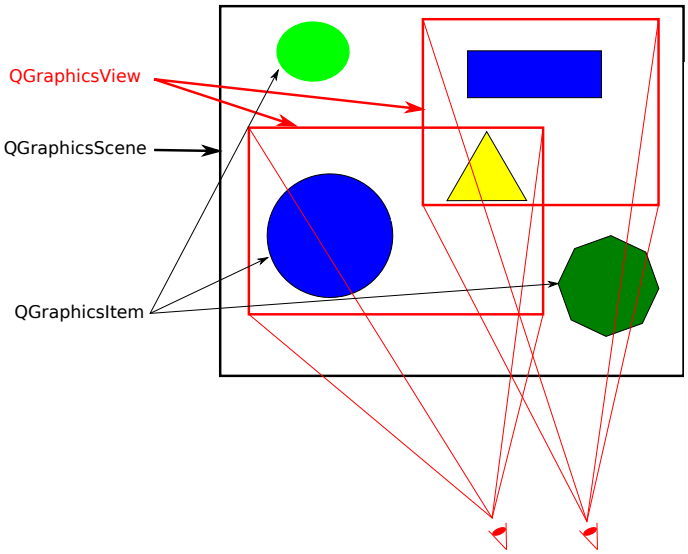
  La vista recibe eventos de entrada del teclado/mouse, y los traslada a la escena.

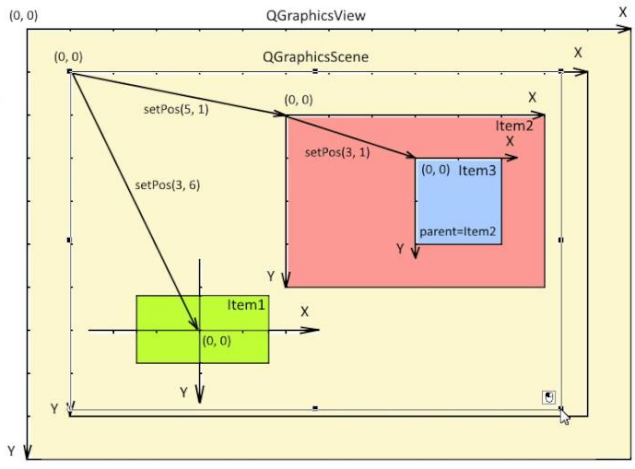
- QGraphicsItem (El Item):

  Representa un grupo de items. Es una clase para el manejo de items gráficos en la escena y proporciona varios items estándar para formas típicas como rectángulos, elipses y textos. También soporta eventos del mouse como mover, soltar, presionar, doble click, sobre y eventos del teclado.

  QGraphicsItem, además soporta dos características importantes en elementos gráficos: Drag and Drop (Soltar y arrastrar) y collision detection.

  Cada objeto QGraphicsItem en la escena soporta rotación, zooming, traslado y el poder cortarlo". `https://ronnyml.wordpress.com/2010/01/12/qgraphicsview/`





[10] Aparece en 2006, con Qt4.2.

2.6.2. Dibujar un Polígono

Programa que dibuja un polígono, relleno con líneas, a partir del número de lados pasado como argumento.

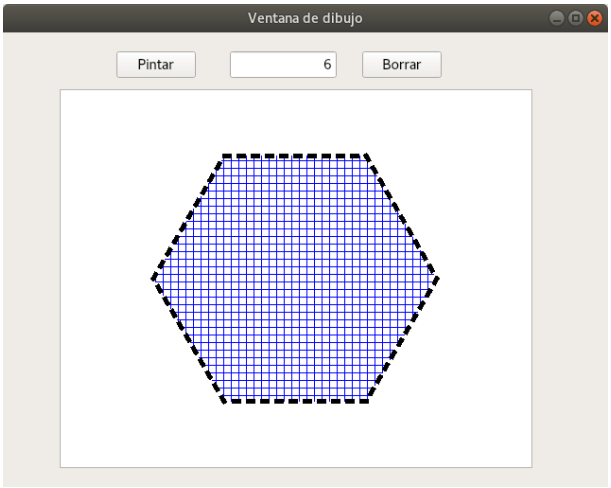
Formulario Creamos un nuevo formulario similar al gráfico de la derecha. En este caso hemos cambiado el nombre de alguno de los objetos. Analiza el código para adecuar los valores del formulario

- Dos botones con los textos que aparecen en la captura.
- Un **lineEdit** para introducir el nº de lados del polígono. De nuevo ajustamos la alineación y controlamos que solo se le puedan pasar números enteros.
- Por último, necesitamos crear una zona para dibujar (**QGraphicsView**) en la que optaremos por seleccionar la propiedad de Antialiasing.

QWidget	
enabled	☑
geometry	[(60, 60), 500 x 400]
X	60
Y	60
Ancho	500
Alto	400
sizePolicy	[Expanding, Expanding, 0, 0]
Política horizontal	Expanding
Política vertical	Expanding
Ajuste horizontal	0
Ajuste vertical	0
minimumSize	0 x 0
Ancho	0
Alto	0

graphicsView : QGraphicsView

Propiedad	Valor
X	0.00
Y	0.00
Ancho	0.00
Alto	0.00
alignment	AlinearCentroH, AlinearCentroV
Horizontal	AlinearCentroH
Vertical	AlinearCentroV
renderHints	Antialiasing|TextAntialiasing
Antialiasing	☑
TextAntialiasing	☑
SmoothPixmapTransform	☐
HighQualityAntialiasing	☐
NonCosmeticDefaultPen	☐
Qt4CompatiblePainting	☐
dragMode	NoDrag
cacheMode	CacheNone



Código ventana.py

```
1 # -*- coding: utf-8 -*-
2
3 """Module implementing Formulario."""
4 from PyQt5 import QtGui, QtCore,  QtWidgets
5 from PyQt5.QtCore import pyqtSlot
6 from PyQt5.QtWidgets import QWidget
7 from PyQt5.QtGui import QPen, QBrush
8 import math
```

```
9
10 from .Ui_ventana import Ui_Formulario
11
12 class Formulario(QWidget, Ui_Formulario):
13
14     """Class documentation goes here."""
15
16     def __init__(self, parent=None):
17         """
18         Constructor.
19
20         @param parent reference to the parent widget
21         @type QWidget
22
23         """
24         super(Formulario, self).__init__(parent)
25         self.setupUi(self)
26         #Creamos la escena
27         self.zonaGrafica.setScene(QtWidgets.QGraphicsScene(self))
28         #Para validar que solo introducimos números enteros
29         self.lineEdit.setValidator(QtGui.QIntValidator())
30
31     def crear_poligono(self, lados,  radio, centro):
32         poligono = QtGui.QPolygonF()
33         angulo = 360/lados             # ángulo para el número de lados
34         for i in range(lados):          # añade los puntos del polígono
35             t = angulo*i + centro
36             x = radio*math.cos(math.radians(t))
37             y = radio*math.sin(math.radians(t))
38             poligono.append(QtCore.QPointF(x, y))
39         return poligono
40
41     @pyqtSlot()
42     def on_pBoton_clicked(self):
43         """Slot documentation goes here."""
44         # TODO: not implemented yet
45         #raise NotImplementedError
46         numeroLados = int(self.lineEdit.text())
47         poligono = self.crear_poligono(numeroLados, 150, 0)
48         # lapiz = QPen(QtGui.QColor('black'), 3, cap=Qt.RoundCap,
           ↪  join=Qt.RoundJoin)
49         # Definimos el tipo de línea y la brocha
50         lapiz = QPen(QtGui.QColor('black'), 5,  style=3)
51         brocha = QBrush(QtGui.QColor('blue'),  style=11)
52         # Los parámetos pen y brush son opcionales
53         self.zonaGrafica.scene().addPolygon(poligono, pen=lapiz,  brush=brocha)
54
55     @pyqtSlot()
56     def on_pushButton_clicked(self):
57         """Slot documentation goes here."""
58         # TODO: not implemented yet
59         #raise NotImplementedError
```

```
60          self.zonaGrafica.scene().clear()
```

Código script principal:

```
from PyQt5 import QtWidgets

from ui.ventana import Formulario

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    ui=Formulario()
    ui.show()
    sys.exit(app.exec_())
```

☺ Veamos algunas cuestiones interesantes de la clase QGraphicsView. Añade al final de las funciones de creación de los elementos gráficos el código

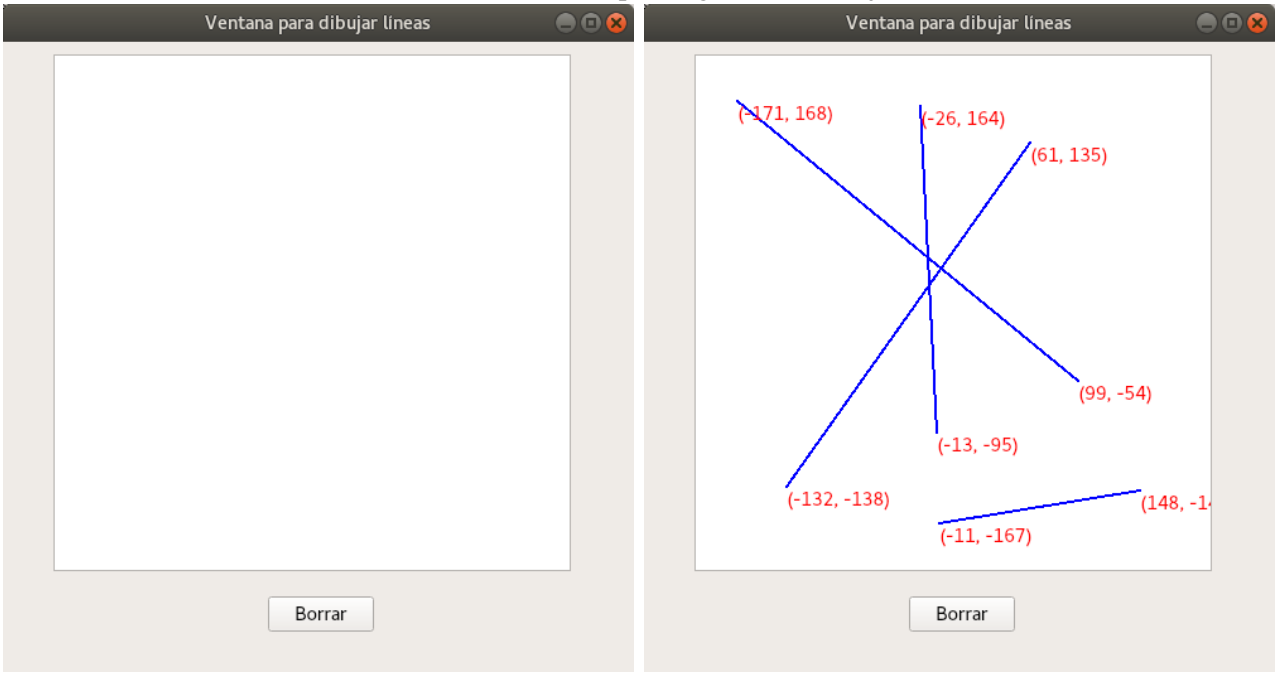
```
for item in self.zonaGrafica.scene().items():
            item.setFlag(QtWidgets.QGraphicsItem.ItemIsMovable)
            item.setFlag(QtWidgets.QGraphicsItem.ItemIsSelectable)
```

¿Qué puedes hacer ahora con los polígonos dibujados?

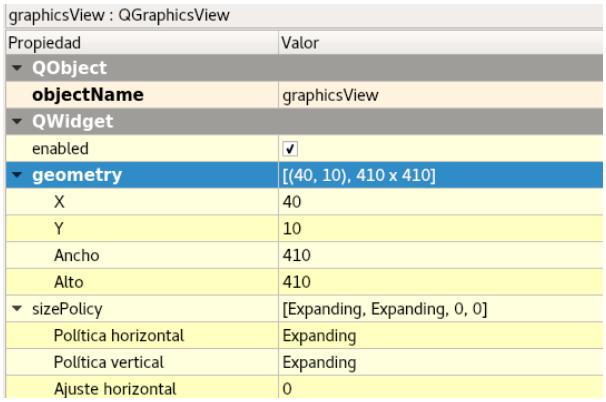
2.6.3. Dibujar líneas y obtener coordenadas.

Programa que permite dibujar líneas, dados el origen y extremo de la línea. También permite obtener las coordenadas de un punto de la zona de dibujo, dibujar la línea con el ratón y practicar con eventos del ratón.

Formulario Crearemos un formulario similar a las capturas gráficas de abajo.

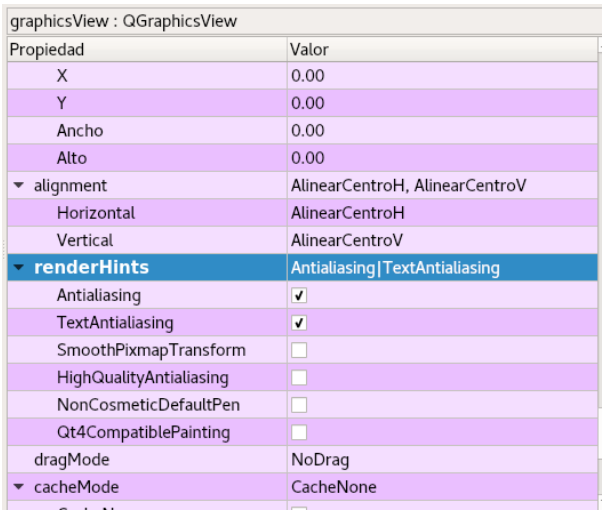


Lo único nuevo a tener en cuenta es que para la zona de dibujo usaremos un QGraphicsView. En este ejemplo su dimensiones son:



graphicsView : QGraphicsView	
Propiedad	Valor
QObject	
objectName	graphicsView
QWidget	
enabled	☑
geometry	[(40, 10), 410 x 410]
X	40
Y	10
Ancho	410
Alto	410
sizePolicy	[Expanding, Expanding, 0, 0]
Política horizontal	Expanding
Política vertical	Expanding
Ajuste horizontal	0

Seleccionamos el Antialiasing para que los gráficos sean de calidad (eliminar el efecto "escalera")



graphicsView : QGraphicsView	
Propiedad	Valor
X	0.00
Y	0.00
Ancho	0.00
Alto	0.00
alignment	AlinearCentroH, AlinearCentroV
Horizontal	AlinearCentroH
Vertical	AlinearCentroV
renderHints	Antialiasing|TextAntialiasing
Antialiasing	☑
TextAntialiasing	☑
SmoothPixmapTransform	☐
HighQualityAntialiasing	☐
NonCosmeticDefaultPen	☐
Qt4CompatiblePainting	☐
dragMode	NoDrag
cacheMode	CacheNone

Código de ventanaprincipal.py

```
1  # -*- coding: utf-8 -*-
2
3  """Module implementing Formulario."""
4  from PyQt5 import QtCore,  QtWidgets,  QtGui
5
6  from PyQt5.QtCore import pyqtSlot
7  from PyQt5.QtWidgets import QWidget
8  #Añadimos para definir el color de las líneas
9  from PyQt5.QtGui import QPen
10
11 from .Ui_ventanaprincipal import Ui_Form
12
13 class Formulario(QWidget, Ui_Form):
14
15     """Class documentation goes here."""
16
17     def __init__(self, parent=None):
18         """
19         Constructor.
20
21         @param parent reference to the parent widget
22         @type QWidget
23
24         """
25         super(Formulario, self).__init__(parent)
26         self.setupUi(self)
27         # Creamos la escena, inicialmente vacía
28         self.graphicsView.setScene(QtWidgets.QGraphicsScene(self))
29         # Se tiene que añadir para que capture los eventos de ratón del
           ↪  graphicsView
30         self.graphicsView.viewport().installEventFilter(self)
31
32     @pyqtSlot()
```

```
    def on_pushButton_clicked(self):
        """Slot documentation goes here."""
        # TODO: not implemented yet
        #raise NotImplementedError
        # Borra la escena, se podría haber programado con QtDesigner
        self.graphicsView.scene().clear()

    def eventFilter(self, source, event):
        #La variable inicio se tiene que definir global para que no se pierdan
        # las coordenadas al volver a entrar en la función
        global inicio
        # Comprobamos los eventos del ratón y actuamos en consecuencia
        if event.type() == QtCore.QEvent.MouseButtonPress:
            if event.button() == QtCore.Qt.LeftButton:
                # Almacenamos en inicio las coordenadas de la pulsación del ratón
                # tomando como referencia las de la escena
                inicio = QtCore.QPointF(self.graphicsView.mapToScene(event.pos()))
        elif event.type() == QtCore.QEvent.MouseButtonRelease:
            fin = QtCore.QPointF(self.graphicsView.mapToScene(event.pos()))
            # Definimos la línea a partir de las coordenadas anteriores
            linea = QtCore.QLineF(inicio, fin)
            # Establecemos el color de las líneas y el grosor, es opcional
            lapiz = QPen(QtGui.QColor('blue'), 2)
            # La variable "lapiz" es opcional, lo mismo que algunas de sus opciones
            #self.graphicsView.scene().addLine(linea)
            self.graphicsView.scene().addLine(linea, pen=lapiz)
            for punto in (inicio, fin):
                # Añadimos las coordenadas de los puntos a la escena
                # Hay que tener en cuenta cómo son los ejes en las escenas
                # Usamos format() para sustituir los valores numéricos en la cadena
                # {0} el primer valor y {1} el segundo
                texto = self.graphicsView.scene().addSimpleText('({0},
                 ↪  {1})'.format(punto.x(), -punto.y()))
                #Establecemos el color del texto en rojo
                texto.setBrush(QtCore.Qt.red)
                # Ponemos el texto en las coordenadas del punto.
                texto.setPos(punto)
        return super(Formulario, self).eventFilter(source, event)
```

Código script principal:

```
from PyQt5 import QtWidgets

from ui.ventanaprincipal import Formulario

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    ui=Formulario()
    ui.show()
    sys.exit(app.exec_())
```

3. Apéndices

3.1. Distribuir nuestros programas

Existen diferentes formas de poder distribuir nuestros programas, una posibilidad es mediante el código fuente creado con eric6, y a través de un terminal ejecutar mediante el comando:

```
python3 script_principal.py
```

Este método tiene el inconveniente de que para poder ejecutarlas, debemos tener una instalación de Python y las librerías necesarias usadas en el programa instaladas en nuestro ordenador.

Es posible distribuir nuestra aplicación Python como un programa autónomo para ejecutarlo en un ordenador que no tenga Python ni las PyQt5 instaladas. Para conseguirlo existen diferentes aplicaciones como PyInstaller o pyqtdeploy.

Nosotros vamos a indicar de forma breve cómo usar PyInstaller. Para instalarlo debemos usar `pip3`[11]. En sistemas Linux basados en paquetes .deb, es en el que lo vamos a comentar, solo tenemos que instalar el paquete adecuado, se trata de `python3-pip`. En otros sistemas, se puede buscar en internet para ver cómo instalarlo. Una vez instalado el paquete anterior, instalamos PyInstaller con el comando:

```
pip3 install pyinstaller
```

Cuando esté instalado en nuestro ordenador, nos situamos (mediante un terminal ya que funciona en modo comando) en la carpeta que contiene el script principal del programa que queremos distribuir

```
cd ruta_carpeta_script_principal
```

y ejecutamos:

```
pyinstaller -F -n programa script_principal.py
```

o

```
pyinstaller -D -n programa script_principal.py
```

para conseguir un ejecutable de nombre `programa` que se situará en la carpeta `dist/` en la que hemos ejecutado los comandos anteriores. La diferencia entre ambas formas de usarlo es que:

1. Con el primer comando conseguimos un solo fichero (de nombre `programa`) que contiene todo lo necesario para poder ejecutar el programa.
2. Con el segundo lo que hacemos es poner en la carpeta anterior una serie de ficheros y carpetas con todo lo necesario para ejecutar la aplicación (el ejecutable tendrá de nombre `programa`). En este caso, para poder ejecutarlo tenemos que usar todos los ficheros de la carpeta creada (por ejemplo, comprimiéndola) si deseamos ejecutar en otra máquina nuestro programa.

Un último comentario, el ejecutable así obtenido solo será compatible con otros sistemas de iguales características (tipo de procesador y sistema operativo). Si lo queremos distribuir en diferentes sistemas, debemos repetir el proceso anterior para cada tipo de ellos.

3.2. Palabras reservadas en Python

Para obtenerlas solo debemos usar

```
>>help("keywords")
```

3.3. Hojas resumen de Python

[11] Sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python

Mémento v1.2.2

Hoja Resumen de Python 3

Documentación oficial de Python
http://docs.python.org/py3k

Tipos Base

enteros, reales, lógicos, textos

```
  int 783    0    -192
float 9.23   0.0   -1.7e-6
 bool True   False
  str "Uno\nDos"   'Pa\'mi'
```

-1.7e-6 → 10^{-6}

"Uno\nDos": nueva línea; 'Pa\'mi': ' escaped

multilínea: """X\tY\tZ
1\t2\t3""" (tabulación)

cadena inmutable, secuencia ordenada de letras

Tipos Contenedores

- secuencia ordenada, índices rápidos, valores repetibles

```
  list [1,5,9]   ["x",11,8.9]   ["texto"]   []
 tuple (1,5,9)   11,"y",7.4     ("texto",)  ()
```

inmutable → tuple; 11,"y",7.4: expresión separada por comas

str: como secuencia ordenada de caracteres

- sin orden previo, llave única, índices rápidos ; llaves = tipos base o tuplas

```
  dict {"llave":"valor"}                    {}
  dict {1:"uno",3:"tres",2:"dos",3.14:"π"}
```

diccionario — *asociaciones llave/valor*

```
  set {"key1","key2"}   {1,9,3,0}   set()
```

Identificadores

para variables, funciones, módulos, clases… nombres

a..zA..Z_ seguidos de a..zA..Z_0..9

- acentos permitidos pero mejor evitarlos
- prohibido usar palabras de python
- discrimina minúsculas/MAYÚSCULAS

```
☺ a toto x7 y_max BigOne
☹ 8y and
```

Asignación de Variables

```
x = 1.2+8+sin(0)
```

x: nombre de variable (identificador); 1.2+8+sin(0): valor o expresión calculada

```
y,z,r = 9.2,-7.6,"bad"
```

y,z,r: nombre de variable; 9.2,-7.6,"bad": contenedor con varios valores (aquí una tupla)

```
x+=3    incrementar
x-=2    decrementar
x=None  «indefinido» valor constante
```

Conversiones

type(*expresión*)

```
int("15")        se puede especificar la base en el 2do parámetro
int(15.56)       trunca la parte decimal (round(15.56) para redondear)
float("-11.24e8")
str(78.3)        y la representación literal ——→ repr("Texto")
```

ver el reverso para mayor control al representar textos

```
bool ——→ use comparadores (con ==, !=, <, >, …), resultado lógico, valor de verdad
list("abc") ——use cada elemento de una secuencia——→ ['a','b','c']
dict([(3,"tres"),(1,"uno")]) ——→ {1:'uno',3:'tres'}
set(["uno","dos"]) ——use cada elemento de una secuencia——→ {'one','dos'}
":".join(['toto','12','pswd']) ——→ 'toto:12:pswd'
```

":": unir textos; ['toto','12','pswd']: secuencia de textos

```
"textos y  espacios".split() ——→ ['textos','y','espacios']
"1,4,8,2".split(",") ——→ ['1','4','8','2']
```

",": separar textos

Indices de secuencias

para listas, tuplas, textos, …

```
len(lst) ——→ 6
```

índices negativos	-6	-5	-4	-3	-2	-1
índices positivos	0	1	2	3	4	5
lst=	[11,	67,	"abc",	3.14,	42,	1968]
corte positivo	0	1	2	3	4	5	6
corte negativo	-6	-5	-4	-3	-2	-1	

acceso individual a los valores [*índice*]

```
lst[1]→67      lst[0]→11    primer valor
lst[-2]→42     lst[-1]→1968 último valor
```

acceso a sub-secuencias via [*inicio corte*:*fin corte*:*pasos*]

```
lst[:-1]→[11,67,"abc",3.14,42]
lst[1:-1]→[67,"abc",3.14,42]
lst[::2]→[11,"abc",42]
lst[:]→[11,67,"abc",3.14,42,1968]
lst[1:3]→[67,"abc"]
lst[-3:-1]→[3.14,42]
lst[:3]→[11,67,"abc"]
lst[4:]→[42,1968]
```

Omitiendo un parámetro de corte → de principio / hasta el fin.

En secuencias mutables, se puede eliminar elementos con `del lst[3:5]` *y modificar asignando* `lst[1:4]=['hop',9]`

Lógica Booleana

Comparadores: < > <= >= == != (≤ ≥ = ≠)

```
a and b   y lógico — ambos simultáneamente
a or b    o lógico — uno, el otro, o ambos
not a     no lógico
True      valor constante verdadero
False     valor constante falso
```

Bloques de Setencias

sentencia madre:
bloque de sentencias 1…
⋮
sentencia madre:
bloque de sentencias 2…
⋮
sentencia siguiente a bloque 1

¡sangría!

Sentencias Condicionales

bloque de sentencias que solo se ejecuta si la condición es verdadera

```
if expresión lógica:
    bloque de sentencias
```

puede tener varios *elif*, *elif*... y solo un *else* al final, ejemplo:

```
if x==42:
    # solo si la expresión lógica x==42 se cumple
    print("realmente verdad")
elif x>0:
    # si no, si la expresión lógica x>0 se cumple
    print("seamos positivos")
elif tamosListos:
    # sino, si la variable lógica tamosListos es verdadera
    print("mira, estamos listos")
else:
    # en todos los otros casos
    print("todo lo demás no fue")
```

Matemáticas

números reales… valores aproximados! — *ángulos en radianes*

Operadores: + - * / // % ** (× ÷ ÷ enteros, resto de ÷, a^b)

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
acos(0.5)→1.0471…
sqrt(81)→9.0   √
log(e**2)→2.0  etc. (cf doc)
```

Sentencia Bucle Condicional

bloque de sentencias que se repite mientras la condición se cumpla

```
while expresión lógica:
    bloque de sentencias
```

```
s = 0
i = 1
```
*inicializaciones **antes** del bucle*

condición con al menos un valor variable (aquí `i`)

```
while i <= 100:
    # sentencias se ejecutan mientras i ≤ 100
    s = s + i**2
    i = i + 1
```
cambiamos el valor condicional

$$s = \sum_{i=1}^{i=100} i^2$$

```
print("suma:",s)
```
resultado computado luego del bucle

¡ cuidado con hacer bucles infinitos !

Control de Bucles

`break` *salir inmediatamente*

`continue` *siguiente iteración*

Sentencia Bucle Iterador

bloque de sentencias ejecutadas para cada ítem de un contenedor o iterador

```
for variable in secuencia:
    bloque de sentencias
```

recorre los **valores** de la secuencia

```
s = "un texto"
cnt = 0
```
*inicializamos **antes** del bucle*

variable de bucle, valor manejado por la sentencia `for`

```
for c in s:
    if c == "t":
        cnt = cnt + 1
print("encontramos",cnt,"'t'")
```
Contamos cantidad de letras `t` en el texto

recorrer un dict/set = recorrer la secuencia de llaves

use cortes para recorrer una subsecuencia

Recorrer lo **índices** de una secuencia
- modificar el ítem correspondiente al índice
- accesar ítemes alrededor del índice (antes/después)

```
lst = [11,18,9,12,23,4,17]
perdidos = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        perdidos.append(val)
        lst[idx] = 15
print("modif:",lst,"-perd:",perdidos)
```
Limita los valores mayores a 15, guarda los valores perdidos.

Recorrer simultáneamente los **índices** y **valores** de una secuencia:

```
for idx,val in enumerate(lst):
```

Entrada / Salida

```
print("v=",3,"cm :",x,",",y+4)
```
ítemes a imprimir: valores literales, variables, expresiones

parámetros de `print`:
- `sep=" "` (separador de ítemes, espacio por omisión)
- `end="\n"` (caracter final, por omisión nueva línea)
- `file=f` (escribir a archivo, por omisión salida estándar)

```
s = input("Instrucciones:")
```
`input` siempre retorna un **texto**, convertir a tipo requerido (revisar *Conversiones* al reverso).

Operaciones sobre Contenedores

`len(c)` → cuenta ítemes

`min(c)` `max(c)` `sum(c)`

`sorted(c)` → *copia* ordenada

*Nota: Para diccionarios y conjuntos, las operaciones son sobre las **llaves**.*

`valor in c` → lógico, operador de membresía `in` (de ausencia, `not in`)

`enumerate(c)` → *iterador* sobre (índice,valor)

*Especial para **contenedores de secuencias** (listas, tuplas, textos) :*

`reversed(c)` → *iterador inverso* `c*5` → duplicados `c+c2` → concadenar

`c.index(val)` → posición `c.count(val)` → cuenta ocurrencias

Generador de Secuencias de Enteros

uso frecuente en bucles iterativos `for`

```
range([inicio,]fin[,paso])
```
por omisión 0, no inclusivo

```
range(5)      → 0 1 2 3 4
range(3,8)    → 3 4 5 6 7
range(2,12,3) → 2 5 8 11
```
`range` retorna un « generador », convertir a lista para ver los valores, por ejemplo:

```
print(list(range(4)))
```

Operaciones sobre Listas

modificar lista original

`lst.append(item)`	añadir ítem al final
`lst.extend(seq)`	añadir secuencia de ítemes al final
`lst.insert(idx,val)`	insertar ítem en un determinado índice
`lst.remove(val)`	elimina el primer ítem con determinado valor
`lst.pop(idx)`	elimina determinado ítem y retorna su valor
`lst.sort()` `lst.reverse()`	ordena / invierte la lista original

Definir Funciones

nombre de función (identificador)

parámetros nombrados

```
def nombfunc(p_x,p_y,p_z):
    """documentación"""
    # bloque de sentencias, calcula result., etc.
    return res
```
valor resultado. si no hay resultado, se retorna: `return None`

parámetros y variables sólo existen *dentro* del bloque y *durante* la llamada a la función (*"caja negra"*)

Invocar Funciones

```
r = nombfunc(3,i+2,2*i)
```
un argumento por parámetro

obtener el valor de retorno (opcional)

Operaciones en Diccionarios

```
d[llave]=valor   d.clear()
d[llave]→valor   del d[llave]
d.update(d2)
```
actualiza/añade asociaciones

```
d.keys()
d.values()
d.items()
```
ver las llaves, valores y asociaciones

```
d.pop(llave)
```

Operaciones en Conjuntos

Operadores:
- `|` → unión (caracter barra vertical)
- `&` → intersección
- `-` `^` → diferencia/diferencia simétrica
- `<` `<=` `>` `>=` → relaciones de inclusión

```
s.update(s2) s.add(valor)
s.remove(llave)
s.discard(llave)
```

Archivos

guardar datos a disco, volver a leerlos

```
f = open("doc.txt","w",encoding="utf8")
```
- **variable** para operaciones
- **nombre** de archivo (+ruta…)
- **modo** de apertura
  - `'r'` lectura
  - `'w'` escritura
  - `'a'` añadir…
- **codificación** de caracteres en archivo: `utf8` `ascii` `latin1` …

consulte funciones en los módulos `os` y `os.path`

escritura

```
f.write("hola")
```
*text file → lee / escribe solo **textos**, convierte convertir al tipo requerido.*

lectura

```
s = f.read(4)
```
vacía si llegamos al fin

si se omite cuantos caracteres, se lee todo el archivo

leer la siguiente línea

```
s = f.readline()
```

```
f.close()
```
no olvider cerrar el archivo al final

Cerrado automático *pytónico* : `with open(…) as f:`

muy común: bucle iterativo para leer las líneas de un archivo de textos

```
for linea in f :
    # bloque que procesa cada línea
```

Formato de Textos

directivas de formato — valores a formatear

```
"model {} {} {}".format(x,y,r) → str
"{selección:formato!conversión}"
```

- **Selection** :
  ```
  2
  x
  0.nombre
  4[llave]
  0[2]
  ```
- Ejemplos:
  ```
  "{:+2.3f}".format(45.7273)
  →'+45.727'
  "{1:>10s}".format(8,"toto")
  →'      toto'
  "{!r}".format("I'm")
  →'"I\'m"'
  ```
- **Formating** :

relleno alineación signo anchomin . precisión~anchomax tipo

`< > ^ =` `+ - espacio` `0` al inicio para rellenar con 0

enteros: `b` binario, `c` caracter, `d` decimal (omisión), `o` octal, `x` or `X` hexa…

reales: `e` o `E` exponencial, `f` or `F` punto fijo, `g` or `G` general (omisión), `%` porcentaje

cadenas: `s` …

- **Conversión** : `s` (texto legible) or `r` (representación literal)

Traducción al Castellano, 2014 – Sebastian Silva

Referencias

[1] Tutorial de Python `http://docs.python.org.ar/tutorial/3/index.html`

[2] Introducción a la programación con Python 3. Andrés Marzal Varó, Isabel Gracia Luengo, Pedro García Sevilla `http://repositori.uji.es/xmlui/bitstream/10234/102653/1/s93.pdf`

[3] `http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos`

[4] `https://es.wikipedia.org/wiki/Python`

[5] PyQt5 Reference Guide `http://pyqt.sourceforge.net/Docs/PyQt5/`

[6] Introducción a la programación con Python `http://www.mclibre.org/consultar/python/`