

# Tema 0 - Programación Básica con Python

## Clase 2

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

## 1 Estructuras de control

- Condicionales
- Bucles o Loops
  - El bucle `while`
  - El bucle `for`

## 1 Estructuras de control

- Condicionales
- Bucles o Loops
  - El bucle `while`
  - El bucle `for`

## 2 Control de errores

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-
- 4 Funciones

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-
- 4 Funciones
- 5 Módulos

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-
- 4 Funciones
- 5 Módulos

# Estructuras de control

En cualquier lenguaje de programación se incluye una serie de estructuras de control para ampliar las posibilidades de ejecución de un programa.

Manejaremos las más comunes que son relativamente sencillas de usar, cuidado siempre de manejar la sintaxis respectiva.



Una sentencia condicional permite evaluar si se cumple cierta condición, es decir, si su valor es `True`, se ejecuta una instrucción, en caso de que el valor de la condición no se cumpla, valor `False`, no se ejecuta la instrucción contenida y se sigue a la siguiente línea de código.

# El condicional `if`

El bloque condicional más simple utiliza la instrucción `if`, a continuación lleva una expresión que debe de evaluarse, como se mencionó antes, el valor de la expresión debe de ser `True` para que se ejecute(n) la(s) instrucción(es) contenidas en el bloque, hay dos puntos (`:`) que identifican el bloque y las instrucciones contenidas deben de estar indentadas, en caso contrario, Python nos devolverá un mensaje de error provocado por la indentación equivocada.

# Ejemplo de if

```
1 a = 10
2 if a > 0:
3     print 'la variable a es positiva'
4     a = a + 1
5
6 print a
```

En el ejemplo asignamos a la variable `a` el valor de 10, en el bloque condicional se evalúa la expresión `a > 10`, y en este caso, el valor que se obtiene es `True`, por tanto, se ejecutan las instrucciones que están contenidas dentro del bloque, que son: 1) mostrar en pantalla la línea `la variable a es positiva`, 2) se incrementa una unidad el valor de `a`. Aquí ya concluyen las instrucciones dentro del condicional, la siguiente instrucción `print` está fuera del bloque, y nos muestra el valor de la variable `a`, que en el ejemplo ahora es 11.

# Cuando la expresión evaluada es falsa

```
1 a = 0
2 if a > 0:
3     print 'la variable a es positiva'
4     a = a + 1
5
6 print a
```

Ahora vemos que el valor de la variable `a` es cero, y al evaluarse la expresión `a > 10`, el resultado es **False**, por tanto NO se ejecuta instrucción alguna que está dentro del bloque, y continua el código hacia la siguiente línea, en el ejemplo, con la instrucción `print`.

# Alternativa 1 para el bloque condicional

En ocasiones necesitaremos hacer algo cuando la expresión que se evalúa en el `if` sea falsa, el programa realice alguna instrucción, es decir, que haya una respuesta en particular para el caso en que obtengamos un valor `False` en la evaluación de la expresión.

Para ello, podemos ocupar la instrucción `else`: que se escribe en el mismo nivel de indentación que la instrucción `if` y lleva también dos puntos (`:`); las instrucciones que están contenidas dentro de `else`: se ejecutarán siempre y cuando la expresión que se evalúa, sea falsa, Python no pide otra expresión para evaluar.

# Condicional if .. else

```
1 a = -2
2 if a > 0:
3     print 'la variable a es positiva'
4     a = a + 1
5 else:
6     print 'la variable a es negativa'
7     print a
```

En el ejemplo vemos que el valor de `a` es `-2`, la expresión que se evalúa en el `if` es `False`, por tanto, no se ejecutan las instrucciones del `if`, sino que se va a ejecutar la instrucción contenida en el `else`:

Si la expresión inicial que se evalúa es `True`, se ejecutan las instrucciones contenidas en el bloque `if`, ya no se ejecuta instrucción alguna del bloque `else`: y continua el programa.

## Alternativa 2 para el bloque condicional

El uso de un bloque `if ... else` nos da oportunidad manejar el código en caso de obtener un valor `False` en la evaluación de la expresión, como ya mencionamos, no se requiere evaluar otra expresión, pero podemos usar una variante del condicional y evaluar una nueva expresión (que sería diferente de la primera) y con ello, nuestro algoritmo gana bastante potencial para responder a nuestras necesidades, y eso lo haremos con el bloque `if ... elif ... else`

# El bloque if..elif..else

```
1 a = 0
2 if a > 0:
3     print 'la variable a es positiva'
4     a = a + 1
5 elif a == 0:
6     print 'la variable a es cero'
7 else:
8     print 'la variable a es negativa'
9
10 print a
```

La primera expresión `a>0` es **False**, por tanto, continua el código hasta la sentencia `elif`:, aquí se evalúa la nueva expresión `a == 0`, que resulta ser **True** y por tanto se ejecuta la instrucción contenida en el bloque `elif`:, saliendo luego del bloque y continua el código, es decir, ya no se revisa el `else`:



# Ejemplo de condicional

```
1 a = input('Introduce el valor de a')
2 if a > 0:
3     print "a es positivo"
4     a = a + 1
5 elif a == 0:
6     print "a es 0"
7 else:
8     print "a es negativo"
9
10 print a:
```

# Uso de los condicionales

Para imprimir formas plurales

```
1 print "Hay %d %s en el banco " % \  
2 (N, (" peso " if N == 1 else " pesos " ))
```

# Uso de los condicionales

Para imprimir formas plurales

```
1 print "Hay %d %s en el banco " % \
2 (N, (" peso " if N == 1 else " pesos " ))
```

Para funciones por tramos

```
1 y = (x if x < 10.0 else (x**2)/10.0)
```

# Uso de los condicionales

Para imprimir formas plurales

```
1 print "Hay %d %s en el banco " % \
2 (N, (" peso " if N == 1 else " pesos " ))
```

Para funciones por tramos

```
1 y = (x if x < 10.0 else (x**2)/10.0)
```

Para evaluar funciones condicionalmente

```
1 def Perim ( radius ): return 2* pi* radius
2 def Area ( radius ): return pi* radius **2
3 v = ( Perim if q == 'perim ' else Area )(1.0)
```

# Bucles o Loops

Un bucle es una sentencia que evalúa inicialmente una condición, en caso de que se cumple (valor True) se ejecuta(n) un conjunto de instrucciones, posteriormente, se revisa el valor de la condición, mientras sea verdadero, las instrucciones se ejecutan nuevamente.

# Tipos de bucles

Hay que considerar dos posibles casos en el manejo de los bucles:

- 1 Cuando conocemos el número de ciclos que van a realizarse.
- 2 Cuando no sabemos cuántas veces se requiere que se repita el ciclo.

En ambos casos se necesita modificar alguna variable y evaluar nuevamente una condición, de tal manera que se cumpla con cierto criterio y concluya el bucle, en caso contrario, tendremos lo que se denomina un *bucle infinito*, es decir, el conjunto de instrucciones se va a repetir indefinidamente, por lo que hay que "cortar" el programa en ejecución.

# El ciclo while

El ciclo while tiene la siguiente sintaxis:

```
while condicion:  
    instruccion1  
    instruccion2  
    ....
```

Las instrucciones contenidas en el bloque se van a ejecutar mientras el valor de la condición sea verdadero, para salir del ciclo, el valor de la condición debe devolver **False**, por lo que es necesario que dentro de este bloque, se realice alguna modificación a la(s) variable(s) contenida(s) en la condición.

# Ejemplo del ciclo while:

```
1 nMax = 5
2 n = 1
3 a = []
4 while n < nMax:
5     a.append(1.0/n)
6     n = n + 1
7 print a
```



## Otro ejemplo del ciclo while:

```
1 from random import  
    random  
2 x = 0.2  
3 while x < 0.6:  
4     x = random()  
5     print x  
6 print "Acabo el bucle"
```

## Otro ejemplo del ciclo while:

```
1 from random import  
    random  
2 x = 0.2  
3 while x < 0.6:  
4     x = random()  
5     print x  
6 print "Acabo el bucle"
```

```
0.452132471948  
0.492677330538  
0.539594612795  
0.0865775779807  
0.0861402799157  
0.96555312462  
Acabó el bucle
```

# El bucle for

La sintaxis del bucle/ciclo for es la siguiente:

```
for variable in lista:  
    instruccion1  
    instruccion2  
    ...
```

El conjunto de instrucciones contenidas dentro del bucle se va a ejecutar mientras no se acabe de recorrer la lista; el valor de la variable, será el elemento de la lista que está siendo tratado en ese momento:

# Ejemplo del ciclo for

```
1 cubos = []  
2 for i in range(7):  
3     ic = i**3  
4     cubos.append(ic)  
5     print i, cubos
```

# Ejemplo del ciclo for

```
1 cubos = []  
2 for i in range(7):  
3     ic = i**3  
4     cubos.append(ic)  
5     print i, cubos
```

```
0 [0]  
1 [0, 1]  
2 [0, 1, 8]  
3 [0, 1, 8, 27]  
4 [0, 1, 8, 27, 64]  
5 [0, 1, 8, 27, 64, 125]  
6 [0, 1, 8, 27, 64, 125, 216]
```

# Algunas cosas más sobre el ciclo for

La instrucción `for` no sólo itera sobre enteros: `for` itera sobre todos los elementos de una *secuencia*, asignando el valor del elemento a la variable, en el ejemplo anterior, la función `range` es sólo una función conveniente que genera una lista de enteros.

```
for i in [ 3, 1, 4, 1, 5, 9, 2, 6, 5, 3 ]:  
    print "Un dígito de pi es", i
```

```
for i in [ 1, 2, 3, 4 ] + [ 3, 2, 1 ]:  
    print i
```

```
for i in " estas son palabras al azar ".split:  
    print i
```

# Ejemplos del ciclo for

```
1 for i in range(10):  
2     print i
```

Al usar la función range, podemos extender el uso del ciclo for

```
1 for i in range(1,10):  
2     print i  
3  
4 for j in range(10,1,-1):  
5     print j  
6  
7 lista = ['adios', 'mundo', 'cruel']  
8 for palabra in lista:  
9     print palabra
```

# Ciclo for más elaborado

El siguiente código usa los elementos de lista y busca la coincidencia exacta del nombre que proporciona el usuario en la línea de comandos, en caso de que no lo encuentre, le avisa al usuario que ese nombre no está en la lista inicial.

```
1 lista = ['Hugo', 'Paco', 'Luis', 'McPato']
2 nombre = raw_input('Teclea un nombre: ')
3
4 for i in range(len(lista)):
5     if lista[i] == nombre:
6         print nombre, ' es el numero ', i + 1, '
          en la lista '
7     break
8 else:
9     print nombre, ' no esta en la lista '
```



# El uso de else en el bloque for

En el ejemplo anterior, vemos que hay una sentencia `else:` al mismo nivel de alineación del ciclo `for`, efectivamente, esta sentencia `else:` no pertenece al condicional `if`, sino al bloque `for`.

La sentencia `else:` la podemos utilizar tanto para los bucles `for` y `while`.

# Instrucciones break y continue

Hay dos instrucciones que permiten "salirse" de un bucle sin necesidad de esperar a que en un ciclo `while`, la expresión que se evalúa, cambie a un valor `False`, y para un ciclo `for`, esperar a que se recorran todos los elementos de la lista.

La instrucción es `break`, veamos en el siguiente ejemplo, cómo se usa.

# Uso de la instrucción break

```
1 for n in range(2, 10):  
2     for x in range(2, n):  
3         if n % x == 0:  
4             print n, ' es igual a ', x, '*', n  
5                 /x  
6                 break  
7     else:  
8         print n, ' es numero primo '
```

Tenemos dos ciclos `for` anidados (hay que tener siempre cuidado con la indentación) y en el segundo ciclo, tenemos un condicional `if` contenido, se evalúa con la función `módulo`, para expresar que un número es producto de otro, y tenemos la instrucción `break`, que no se espera a que se complete el recorrido de los elementos de la lista con índice `x`, que va desde 2 hasta `n`, y continua con un incremento en el contador `n` del primer ciclo `for`, en caso de que no haya un residuo igual a cero, entonces el número es primo, aquí nos apoyamos en la sentencia `else` del ciclo `for`.

# Uso de la instrucción continue

La instrucción continue, lo que hace, es dar paso a la siguiente iteración del ciclo, veamos el ejemplo:

```
1 for num in range(2,10):  
2     if num % 2 == 0:  
3         print 'Encontre un numero par ', num  
4         continue  
5     print 'Encontre un numero ', num
```

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-
- 4 Funciones
- 5 Módulos

# Control de errores

Cuando comenzamos a programar, nos podemos encontrar con mensajes de error al momento de ejecutar el programa, siendo las causas más comunes:

- errores de dedo, escribiendo incorrectamente una instrucción, sentencia, variable o constante.

# Control de errores

Cuando comenzamos a programar, nos podemos encontrar con mensajes de error al momento de ejecutar el programa, siendo las causas más comunes:

- errores de dedo, escribiendo incorrectamente una instrucción, sentencia, variable o constante.
- errores al momento de introducir los datos, por ejemplo, si el valor que se debe de ingresar es 123.45, y si nosotros tecleamos 1234.5, el resultado ya se considera un error.



# Control de errores

Cuando comenzamos a programar, nos podemos encontrar con mensajes de error al momento de ejecutar el programa, siendo las causas más comunes:

- errores de dedo, escribiendo incorrectamente una instrucción, sentencia, variable o constante.
- errores al momento de introducir los datos, por ejemplo, si el valor que se debe de ingresar es 123.45, y si nosotros tecleamos 1234.5, el resultado ya se considera un error.
- errores que se muestran en tiempo de ejecución, es decir, todo está bien escrito y los datos están bien introducidos, pero hay un error debido a la lógica del programa o del método utilizado, ejemplo: división entre cero.

# Manejo de errores

```
c = 12.0/0.0
```

# Manejo de errores

```
c = 12.0/0.0
```

```
Traceback (most recent call last):  
File '<pyshell#0>', line 1, in ?  
c = 12.0/0.0  
ZeroDivisionError: float division
```

# Manejo de errores

```
c = 12.0/0.0
```

```
Traceback (most recent call last):  
File '<pyshell#0>', line 1, in ?  
c = 12.0/0.0  
ZeroDivisionError: float division
```

```
try:  
    c = 12.0/0.0  
except ZeroDivisionError:  
    print 'Division entre cero'
```

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-
- 4 Funciones
- 5 Módulos

# Por qué usar una interfaz de desarrollo?

Para el desarrollo de problemas científicos con Python, hemos visto y recurrido a la sintaxis y estructuras de control propias del lenguaje, así como ejecutar un programa (o script) desde la terminal.

Cada vez, nuestros códigos aumentarán de tamaño y el riesgo también se incrementa, si omitimos alguna instrucción o dejamos un bloque sin indentar, no cerramos un paréntesis, etc. Además, la interacción con varias terminales en linux, hace más complicado el orden y control.

Tenemos disponibles bajo licencia GNU, varios IDEs para programar con Python. La ventaja es que precisamente integran mucho el trabajo que tenemos que hacer a mano, resalta con colores las instrucciones, nos ofrece ventanas para visualizar los resultados, sin necesidad de tener abiertas varias terminales.

# Usando gEdit

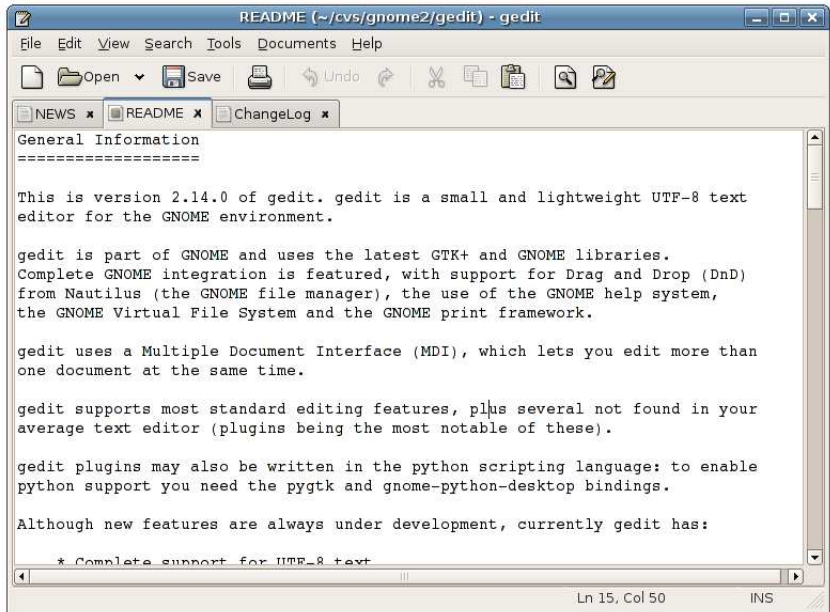
`gedit` es un editor de textos compatible con UTF-8 para GNU/Linux, Mac OS X y Microsoft Windows. Diseñado como un editor de textos de propósito general, `gedit` enfatiza la simplicidad y facilidad de uso. Incluye herramientas para la edición de código fuente y textos estructurados, como lenguajes de marcado.

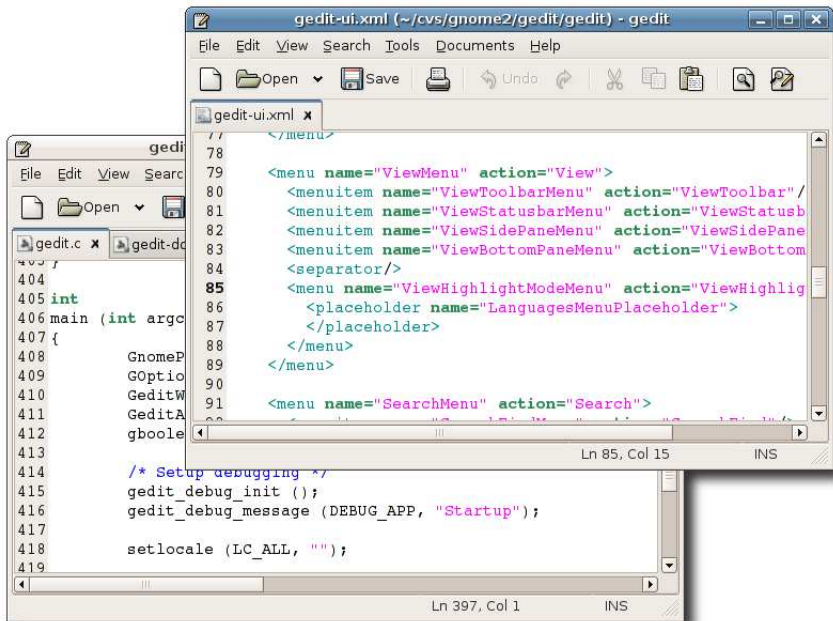
Es el editor predeterminado de GNOME.

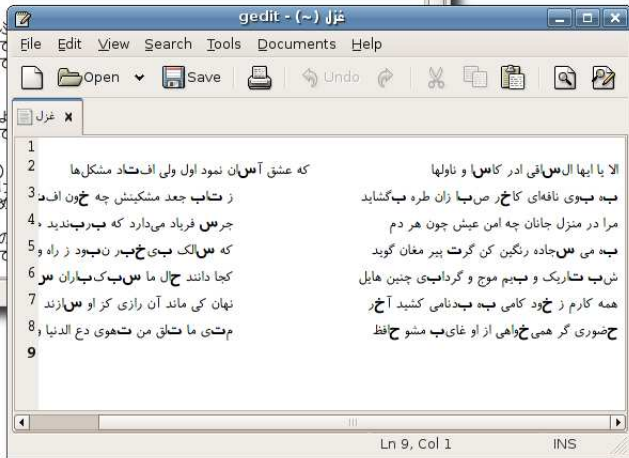
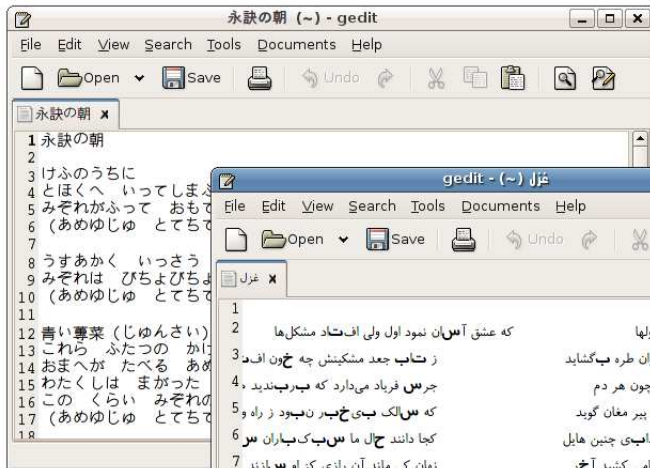
Distribuido bajo las condiciones de la licencia GPL, `gedit` es software libre.

<https://wiki.gnome.org/Apps/Gedit>

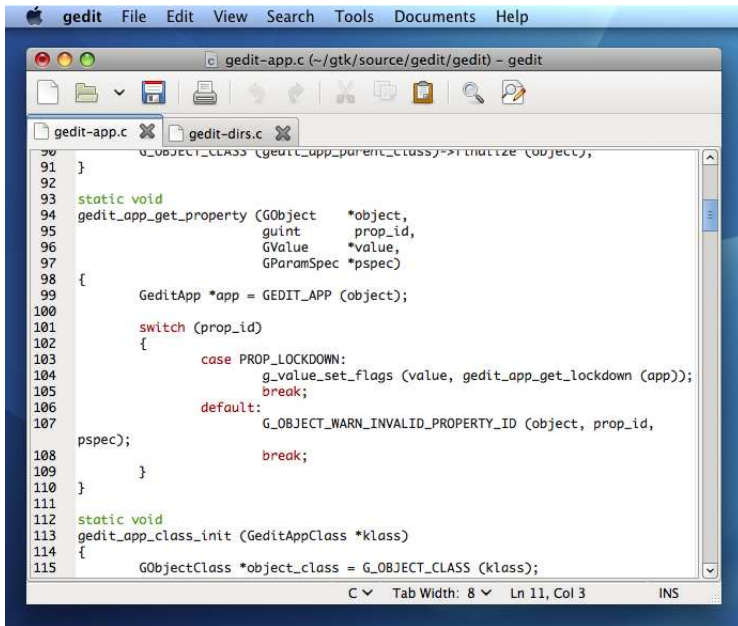








```
iso_639.xml (Vista (C:) \Program Files\gedit\share\xml\iso-codes) - gedit
File Edit View Search Tools Documents Help
New Open Save Print... Undo Redo Cut Copy Paste Find Replace
README iso_639.xml
Source: <http://www.loc.gov/standards/iso639-2/>
-->
<!DOCTYPE iso_639_entries [
  <!ELEMENT iso_639_entries (iso_639_entry+)>
  <!ELEMENT iso_639_entry EMPTY>
  <!ATTLIST iso_639_entry
    iso_639_2B_code    CDATA    #REQUIRED
    iso_639_2T_code    CDATA    #REQUIRED
    iso_639_1_code     CDATA    #IMPLIED
    name               CDATA    #REQUIRED
  >
]>
<iso_639_entries>
  <iso_639_entry
    iso_639_2B_code="aar"
    iso_639_2T_code="aar"
    iso_639_1_code="aa"
    name="Afar" />
  <iso_639_entry
    iso_639_2B_code="abk"
    iso_639_2T_code="abk"
    iso_639_1_code="ab"
    name="Abkhazian" />
XML Tab Width: 8 Ln 1, Col 2 INS
```



```
90     G_OBJECT_CLASS (gedit_app_parent_class)->finalize (object),
91 }
92
93 static void
94 gedit_app_get_property (GObject      *object,
95                        guint         prop_id,
96                        GValue       *value,
97                        GParamSpec   *pspec)
98 {
99     GeditApp *app = GEDIT_APP (object);
100
101     switch (prop_id)
102     {
103     case PROP_LOCKDOWN:
104         g_value_set_flags (value, gedit_app_get_lockdown (app));
105         break;
106     default:
107         G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id,
108         pspec);
109         break;
110     }
111 }
112
113 static void
114 gedit_app_class_init (GeditAppClass *klass)
115 {
116     GObjectClass *object_class = G_OBJECT_CLASS (klass);
```

# El entorno Spyder2

Spyder es un entorno de desarrollo integrado para el lenguaje Python con pruebas interactivas y funciones avanzadas de depuración, introspección y edición.

Spyder permite trabajar fácilmente con las mejores herramientas de la pila científica de Python en un entorno sencillo y potente.

<https://code.google.com/p/spyderlib/>

Estas son algunas de las características clave de Spyder:

- 1 Cuadro de diálogo de administración de PYTHONPATH como de MATLAB (funciona con todas las consolas)

Estas son algunas de las características clave de Spyder:

- 1 Cuadro de diálogo de administración de PYTHONPATH como de MATLAB (funciona con todas las consolas)
- 2 Editor de variables de entorno de usuario actual.



Estas son algunas de las características clave de Spyder:

- 1 Cuadro de diálogo de administración de PYTHONPATH como de MATLAB (funciona con todas las consolas)
- 2 Editor de variables de entorno de usuario actual.
- 3 Enlaces directos a la documentación (Python, Matplotlib, NumPy, Spicy, etc.)

Estas son algunas de las características clave de Spyder:

- 1 Cuadro de diálogo de administración de PYTHONPATH como de MATLAB (funciona con todas las consolas)
- 2 Editor de variables de entorno de usuario actual.
- 3 Enlaces directos a la documentación (Python, Matplotlib, NumPy, Spicy, etc.)
- 4 Enlace directo al lanzador de Python(x,y)

Estas son algunas de las características clave de Spyder:

- 1 Cuadro de diálogo de administración de PYTHONPATH como de MATLAB (funciona con todas las consolas)
- 2 Editor de variables de entorno de usuario actual.
- 3 Enlaces directos a la documentación (Python, Matplotlib, NumPy, Spicy, etc.)
- 4 Enlace directo al lanzador de Python(x,y)
- 5 Enlaces directos a QtDesigner, QtLinguist y QtAssistant (documentación de Qt)

Spyder

File Edit Search Source Run Tools View ?

Editor - C:\Documents and Settings\carlos\Mis documentos\Python\montecarlo\_pi.py

Interpolation.py montecarlo\_pi.py

```
1#!/usr/bin/env python
2# -*- coding: utf-8 -*-
3"""Simple generation of pi via MonteCarlo integration.
4Taken from the Py4Science Workbook.
5"""
6import math
7import random
8import numpy as np
9from scipy import weave
10
11def v1(n = 100000):
12    """Approximate pi via monte carlo integration"""
13    rand = random.random
14    sqrt = math.sqrt
15    sm = 0.0
16    for i in xrange(n):
17        sm += sqrt(1.0-rand())**2
18    return 4.0*sm/n
19
20def v2(n = 100000):
21    """Implement v1 above using weave for the C call"""
22    support = "#include <stdlib.h>"
23    code = """
24double sm;
25float rnd;
26srand(1); // seed random number generator
27sm = 0.0;
28for(int i=0;i<n;++i) {
29    rnd = rand()/(RAND_MAX+1.0);
30    sm += sqrt(1.0-rnd*rnd);
31}
```

Object inspector

Source Console Object numpy.mean

**mean(a, axis=None, dtype=None, out=None)**  
Function of numpy.core.fromnumeric module

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over flattened array by default, otherwise over the specified axis. float64 intermediate and return values are used for integer inputs.

**Parameters**

**a** : array\_like  
Array containing numbers whose mean is desired. If a is not array, a conversion is attempted.

**axis** : int, optional  
Axis along which the means are computed. The default is to compute the mean of the flattened array.

Object inspector Variable explorer File explorer

Console

IPython 1 00:04

```
In [2]: sin([1,2,3])
Out[2]: array([ 0.84147098,  0.90929743,  0.14117657])

In [3]:
```

Spyder

File Edit Search Source Run Tools View ?

Editor - /home/carlos/Escritorio/montecarlo\_pi.py

Interpolation.py montecarlo\_pi.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 """Simple generation of pi via MonteCarlo integration.
4 Taken from the Py4Science Workbook.
5 """
6 import math
7 import random
8 import numpy as np
9 from scipy import weave
10
11 def v1(n = 100000):
12     """Approximate pi via monte carlo integration"""
13     rand = random.random
14     sqrt = math.sqrt
15     sm = 0.0
16     for i in xrange(n):
17         sm += sqrt(1.0-rand()*2)
18     return 4.0*sm/n
19
20 def v2(n = 100000):
21     """Implement v1 above using weave for the C call"""
22     support = "#include <stdlib.h>"
23     code = """
24 double sm;
25 float rnd;
26 srand(1); // seed random number generator
27 sm = 0.0;
28 for(int i=0;i<n;++i) {
29     rnd = rand()/(RAND_MAX+1.0);
30     sm += sqrt(1.0-rnd*rnd);
31 }
```

Object inspector

Source Console Object mean

**mean(a, axis=None, dtype=None, out=None)**  
Function of numpy.core.fromnumeric module

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. float64 intermediate and return values are used for integer inputs.

**Parameters**

- a** : array\_like  
Array containing numbers whose mean is desired. If not an array, a conversion is attempted.

Object inspector Variable explorer File explorer

Console

IPython 1 IPython 2 00:01:1

```
In [2]: sin([1,2,3])
Out[2]: array([ 0.84147098,  0.90929743,  0.14112001])

In [3]: |
```

Console History log

# Otros IDEs para Python

Pueden probar los IDE mencionados, pero también es importante señalar que hay otros entornos, cada uno con ciertas características que lo hacen particular. El punto es que si trabajan con uno, lo puedan explotar al máximo.

Una lista de otros IDEs para programar con Python, la pueden consultar en:

<https://wiki.python.org/moin/PythonEditors>

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-
- 4 Funciones
- 5 Módulos

Con lo que hemos revisado sobre Python, tenemos elementos para iniciar la solución de problemas, una manera particular de agrupar un conjunto de instrucciones, es a través de funciones.

Las funciones intrínsecas de cualquier lenguaje son pocas, pero podemos extenderlas con funciones definidas por el usuario.



# Estructura de una función

La estructura de una función en Python es la siguiente:

```
def nombre_funcion(parametro1, parametro2, .  
    conjunto de instrucciones  
    return valores_devueltos
```

donde parametro1, parametro2 son los parámetros. Un parámetro puede ser cualquier objeto de Python, incluyendo una función.

Los parámetros pueden darse por defecto, por lo que en la función son opcionales. Si no se utiliza la instrucción return, la función devuelve un objeto null

# Ejemplo

```
1 def cuadrados(a):  
2     for i in range(len(a)):  
3         a[i] = a[i]**2  
4  
5 a = [1, 2, 3, 4]  
6 cuadrados(a)  
7 print a
```

# Paso de argumentos

Para que una función sea en verdad útil (y reutilizable), es necesario que podamos pasarle entradas. Los nombres de las entradas (o argumentos) que requiere una función se declaran a continuación del nombre en `def` (siempre entre paréntesis)

```
def FuncionSuma (x, y):  
    return x + y  
  
print FuncionSuma (5, 3)  
print FuncionSuma (7, 42.0)  
print FuncionSuma (" hola ", " mundo ")
```

## Nota:

- Nunca se mencionan los tipos de datos de x e y, ni el tipo de datos que devuelve FuncionSuma.
- Los argumentos y el valor devuelto son, tal como las variables, simples etiquetas a zonas de memoria.

# Validar el tipo de dato

A veces se va a requerir la validación del tipo de dato de manera explícita (aunque no es para nada pitónico!).

Un truco más o menos claro para emular lenguajes estáticos es usar la función `type` y la instrucción `assert`.

```
def SumaEnteros (x, y):  
    assert type (x) == int  
    assert type (y) == int  
    return x+y
```

```
print SumaEnteros (5, 3) # -> 8  
print SumaEnteros (7, 42.0) # -> AssertionError
```

La instrucción `assert` actúa como filtro si la expresión que le sigue es verdadera, pero falla con `AssertionError` si es falsa.

Es más común usar `try` y `except` para "atrapar" el error si los tipos no son los adecuados.

# Paso de argumentos con nombre

Si la función que definimos tiene muchos argumentos, es fácil olvidar el orden en que fueron declarados.

Como un argumento no lleva asociado un tipo, Python no tiene manera de saber que los argumentos están cambiados.

Para evitar este tipo de errores, hay una manera de llamar a una función pasando los argumentos en cualquier orden arbitrario: se pasan usando el nombre usado en la declaración.

```
def Prueba (a, b, c):  
    # %r formatea automaticamente cualquier valor  
    print "a= %r, b= %r, c= %r" % (a, b, c)
```

```
Prueba (1, 2, 3)  
Prueba (b=3, a=2, c=1)
```

```
a=1, b=2, c=3  
a=2, b=3, c=1
```



# Argumentos con valores por omisión

Para hacer que algunos argumentos sean opcionales, se les da valores por omisión en el momento de declararlos:

```
from math import sqrt
# argumento v es requerido , c es opcional
# c toma el valor 3.0e8 por omision
def Gamma (v, c = 3.0e+8):
    return sqrt (1.0 -(v/c)**2)

print Gamma (0.1 , 1.0)
print Gamma (1.e+7) # usa c = 3.0e+8
```

# Regresando varios valores en una función

Para hacer que una función devuelva más de un valor, en lenguajes como Fortran, C o C++, lo que se hace es definir argumentos de entrada y argumentos de salida.

Para devolver múltiples valores en Python, lo usual es devolver los valores "empaquetados en una tupla:

```
from math import atan , sqrt
```

```
def ModuloArgumento (x, y):  
    norm = sqrt (x**2 + y**2)  
    arg = atan2 (y, x)  
    return (norm,arg)
```

```
n, a = ModuloArgumento (3.0,4.0)  
print " Modulo es:", n  
print " Argumento es:", a
```

# Número variable de argumentos

¿Cómo le hacemos para que una función acepte un número no prefijado de argumentos?

Es posible pasar una lista o tupla, pero Python ofrece una mejor solución:

```
def atan(*args ):
    # args es una tupla de argumentos
    if len(args) == 1:
        return math.atan(args[0])
    else :
        return math.atan2(args[0],args[1])
```

```
print atan (0.2) # 0.19739
print atan (2.0,10.0) # 0.19739
print atan (-2.0,-10.0) # -2.94419
```

# Cálculo de la serie de Fibonacci

La sucesión fue descrita por Fibonacci como la solución a un problema de la cría de conejos:

"Cierta hombre tenía una pareja de conejos juntos en un lugar cerrado y uno desea saber cuántos son creados a partir de este par en un año cuando es su naturaleza parir otro par en un simple mes, y en el segundo mes los nacidos parir también"

# Cálculo de la serie de Fibonacci

La sucesión fue descrita por Fibonacci como la solución a un problema de la cría de conejos:

"Cierta hombre tenía una pareja de conejos juntos en un lugar cerrado y uno desea saber cuántos son creados a partir de este par en un año cuando es su naturaleza parir otro par en un simple mes, y en el segundo mes los nacidos parir también"

cómo le hacemos?

# Propuesta de código

```
1 a, b = 0, 1
2 while b < 10:
3     print b
4     a, b = b, a+b
```

# Propuesta de código

```
1 a, b = 0, 1
2 while b < 10:
3     print b
4     a, b = b, a+b
```

```
1 a, b= 0, 1
2 while b < 1000:
3     print b,
4     a, b = b, a+b
```

- 1 Estructuras de control
  - Condicionales
  - Bucles o Loops
    - El bucle `while`
    - El bucle `for`
- 2 Control de errores
- 3 Uso de Intefases de Desarrollo -IDE-
- 4 Funciones
- 5 Módulos



Es una buena práctica almacenar las funciones en módulos. Un módulo es un archivo en donde se dejan las funciones, el nombre del módulo es el nombre del archivo.

Un módulo se carga al programa con la instrucción

```
from nombre_modulo import *
```

Python incluye un número grande de módulos que contienen funciones y métodos para varias tareas. La gran ventaja de los módulos es que están disponibles en internet y se pueden descargar, dependiendo de la tarea que se requiera atender.

Muchas funciones matemáticas no se pueden llamar directo del intérprete, pero para ello existe el módulo `math`.

Hay tres diferentes maneras en las que se puede llamar y utilizar las funciones de un módulo.

```
from math import *
```

De esta manera, se importan todas las funciones definidas en el módulo `math`, siendo quizá un gasto innecesario de recursos, pero también generar conflictos con definiciones cargadas de otros módulos.

```
from math import func1, func2,...
```

```
from math import func1, func2,...
```

```
>>> from math import log,sin  
>>> print log(sin(0.5))  
-0.735166686385
```

El tercer método que es el más usado en programación, es tener disponible el módulo:

```
import math
```

Las funciones en el módulo se pueden usar con el nombre del módulo como prefijo:

```
>>> import math  
>>> print math.log(math.sin(0.5))  
-0.735166686385
```

# Contenido del módulo math

Podemos ver el contenido de un módulo con la instrucción:

```
>>> import math  
>>> dir(math)
```

```
['__doc__', '__name__', '__package__', 'acos',  
'acosh', 'asin', 'asinh', 'atan', 'atan2',  
'atanh', 'ceil', 'copysign', 'cos', 'cosh',  
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',  
'fabs', 'factorial', 'floor', 'fmod', 'frexp',  
'fsum', 'gamma', 'hypot', 'isinf', 'isnan',  
'ldexp', 'lgamma', 'log', 'log10', 'log1p',  
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',  
'sqrt', 'tan', 'tanh', 'trunc']
```