Tema 0 - Programación básica con `python 3` Clase 3

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

Contenido

Contenido

Contenido

Contenido

Contenido

Por qué usar una interfaz de desarrollo?

Para el desarrollo de problemas científicos con Python, hemos visto y recurrido a la sintaxis y estrucutras 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 identar, no cerramos un paréntesis, etc. Además, la interacción con varias terminales en linux, hace más complicado el orden y control.

Existen varios IDEs para programar con `python` bajo licencia GNU. La ventaja es que precisamente integra mucho del 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.

¿Cuál es el mejor IDE? Esta respuesta está en función de tus gustos, del modo que prefieras trabajar, ya que algunos editores son muy propios de la programación desde terminal, otros que tienen una interfaz gráfica que facilita el trabajo.

Entornos de desarrollo para python.

- Eclipse + pydev
- Netbeans
- SPE IDE
- Eric
- Komodo Edit
- Geany

- WingIDE
- IDLE
- PyCharm
- NINJA-IDE
- Spyder

Editores de texto avanzados.

- Vim
- Emacs
- Gedit
- Kate

- Marave
- Textmate
- Scribes
- Notepad++

El entorno Spyder3

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.

Para consultar la documentación del programa, puedes visitar la siguiente página:
`https://pythonhosted.org/spyder/`

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)

Contenido

Funciones en `python`.

Una función es un bloque de código organizado y reutilizable que se utiliza para realizar una sola acción/tarea. Las funciones proporcionan una modularidad para nuestra objetivo de resolver problemas de cómputo científico, con la ventaja de contar con un alto grado de reutilización de código.

Como ya sabes, `python` ofrece muchas funciones integradas -intrísecas- como `print ()`, etc., pero también puedes crear tus propias funciones. Estas funciones se denominan funciones definidas por el usuario.

Definición de una función.

Se pueden definir funciones para proporcionar la operacionalidad que requerimos. Aquí hay que considerar las siguientes *reglas simples para definir una función* en `python`.

1. Un bloque de función comienzan con la palabra clave `def` seguido por el nombre de la función y los paréntesis ().

Definición de una función.

Se pueden definir funciones para proporcionar la operacionalidad que requerimos. Aquí hay que considerar las siguientes *reglas simples para definir una función* en `python`.

1. Un bloque de función comienzan con la palabra clave `def` seguido por el nombre de la función y los paréntesis ().
2. Cualquier parámetro o argumento de entrada debe colocarse dentro de estos paréntesis.

Definición de una función.

Se pueden definir funciones para proporcionar la operacionalidad que requerimos. Aquí hay que considerar las siguientes *reglas simples para definir una función* en `python`.

1. Un bloque de función comienzan con la palabra clave `def` seguido por el nombre de la función y los paréntesis ().
2. Cualquier parámetro o argumento de entrada debe colocarse dentro de estos paréntesis.
3. La primera declaración de una función puede ser una instrucción opcional: una cadena de documentación de la función o `docstring`.

Definición de una función.

Se pueden definir funciones para proporcionar la operacionalidad que requerimos. Aquí hay que considerar las siguientes *reglas simples para definir una función* en `python`.

1. Un bloque de función comienzan con la palabra clave `def` seguido por el nombre de la función y los paréntesis ().
2. Cualquier parámetro o argumento de entrada debe colocarse dentro de estos paréntesis.
3. La primera declaración de una función puede ser una instrucción opcional: una cadena de documentación de la función o `docstring`.
4. El bloque de código dentro de cada función comienza con dos puntos (:) y está identado.

Definición de una función.

Se pueden definir funciones para proporcionar la operacionalidad que requerimos. Aquí hay que considerar las siguientes *reglas simples para definir una función* en `python`.

1. Un bloque de función comienzan con la palabra clave `def` seguido por el nombre de la función y los paréntesis ().
2. Cualquier parámetro o argumento de entrada debe colocarse dentro de estos paréntesis.
3. La primera declaración de una función puede ser una instrucción opcional: una cadena de documentación de la función o `docstring`.
4. El bloque de código dentro de cada función comienza con dos puntos (:) y está identado.
5. La sentencia `return [expresión]` "sale" de una función, devolviendo una expresión. Una sentencia `return` sin argumentos es igual que `return None`.

Sintaxis de una función en python.

```
1 def nombrefuncion( [par1][, par2] ..[,parn] ):
2     'cadena de documentacion o docstring'
3     instrucciones de la funcion
4     ...
5     ...
6     return [expresion]
```

De forma predeterminada, los parámetros tienen un comportamiento posicional y es necesario ingresarlos en el mismo orden en que se definieron.

Llamar a una función.

Al definir una función se le da un nombre, se especifica(n) el(los)parámetro(s) que se van a incluir en la función y se estructura en bloques el código que se va a ejecutar.

Una vez finalizada la estructura básica de una función, se ejecuta llamándola desde otra función o directamente desde el `python`.

Ejemplo de la llamada de una función.

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

Ejemplo de la llamada de una función.

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

Se define inicialmente la función `cuadrados`, posteriormente en el código se define el objeto *a* que es una lista, se manda llamar la función dando como argumento la lista *a* y luego se visualiza el contenido que regresa la función. Nótese que `return` no tiene argumento, por lo que la función sólo ejecuta las instrucciones contenidas en la función.

Paso de arguementos: por referencia vs valor.

Todos los parámetros (argumentos) en `python` **se pasan por referencia**.

Esto significa que si cambia lo que un parámetro se refiere dentro de una función, el cambio también se refleja de nuevo en la función de llamada.

Ejemplo de paso de argumentos por referencia.

```
1  # Se define la funcion
2  def cambiame( milista ):
3      print ('Valores dentro de la funcion antes del
           cambio: ', milista)
4
5      milista[2]=50
6
7      print ('Valores dentro de la funcion luego del
           cambio: ', milista)
8      return
9
10 # Llamamos a la funcion
11 milista = [10,20,30]
12 cambiame( milista )
13 print ('Valores fuera de la funcion: ', milista)
```

Otro ejemplo de paso por referencia.

```
1  # Se define la funcion
2  def cambiame2( milista ):
3      milista = [1,2,3,4] # Asigna una nueva
           referencia a milista
4      print ('Valores dentro de la funcion antes del
           cambio: ', milista)
5      return
6
7  # Llamamos a la funcion
8  milista = [10,20,30]
9  cambiame( milista )
10 print ('Valores fuera de la funcion: ', milista)
```

Argumentos de la función.

Se puede llamar a una función utilizando los siguientes tipos de argumentos formales:

- Argumentos requeridos.
- Argumentos por nombre.
- Argumentos por defecto.
- Argumentos de longitud variable.

Argumentos requeridos.

Los argumentos requeridos son los argumentos pasados a una función en orden posicional correcto. Aquí, el número de argumentos en la llamada de función debe coincidir exactamente con la definición de la función.

Para llamar a la función `cuadrados ()`,se necesita pasar un argumento, de lo contrario da un error de sintaxis:

Error en la llamada de una función.

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

Error en la llamada de una función.

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

```
Traceback (most recent call last):

  "funcioncuadrados.py", line 15, in <module>
    cuadrados()

TypeError: cuadrados() missing 1 required positional
    argument: 'a'
```

Argumentos por nombre.

Cuando se utilizan argumentos por nombre en una llamada de función, en la instrucción donde se llama a la función, se identifican los argumentos por el nombre del parámetro.

Esto permite saltar argumentos o colocarlos fuera de orden porque el intérprete de `python` puede usar los nombres proporcionados para emparejar los valores con los parámetros.

Ejemplo de paso de argumentos por nombre.

```
1 # Se define la funcion
2 def imprimecadena(micadena):
3     print (micadena)
4     return
5
6 # Llamamos a la funcion
7 imprimecadena(micadena = 'Hola Mundo!!')
```

Otro ejemplo de paso de argumentos por nombre.

```
1 # Se define la funcion
2 def imprimeinfo(nombre, edad):
3     print ('Nombre: ', nombre)
4     print ('Edad: ', edad)
5     return
6
7 # Llamamos a la funcion
8 imprimeinfo(edad=15, nombre='Blanca Nieves')
```

Argumentos por defecto.

Un argumento por defecto es un argumento que asume un valor predeterminado si no se proporciona un valor en la llamada de función para ese argumento.

El siguiente ejemplo da una idea sobre los argumentos por defecto, imprime la edad predeterminada si no se proporciona.

Ejemplo de paso de argumentos por defecto.

```
1  # Se define la funcion
2  def imprimeinfo(nombre, edad = 30):
3      print ('Nombre: ', nombre)
4      print ('Edad: ', edad)
5      return
6
7  # Llamamos a la funcion
8  imprimeinfo(edad=15, nombre='Blanca Nieves')
9
10 imprimeinfo(nombre = 'Lara Croft')
```

Argumentos de longitud variable.

Es posible que necesitemos usar una función con más argumentos de los que especificamos al definir la función.

Estos argumentos se denominan argumentos de longitud variable y no se nombran en la definición de función, a diferencia de los argumentos requeridos y por nombre. La sintaxis para una función con argumentos variables es:

Argumentos de longitud variable.

Es posible que necesitemos usar una función con más argumentos de los que especificamos al definir la función.

Estos argumentos se denominan argumentos de longitud variable y no se nombran en la definición de función, a diferencia de los argumentos requeridos y por nombre. La sintaxis para una función con argumentos variables es:

```
1 def nombrefuncion([args_formal,] *tupla_args ):
2     bloque de instrucciones
3     return [expresion]
```

Un asterisco ($*$) se coloca antes del nombre de variable que contiene los valores de todos los argumentos de variable por nombre.

La tupla permanece vacía si no se especifican argumentos adicionales durante la llamada de función.

Ejemplo de una función con argumentos variables.

```
1  # Se define la funcion
2  def presentainfo( arg1, *vartupla ):
3      print ('La salida es: ')
4      print (arg1)
5      for var in vartupla:
6          print (var)
7      return
8
9  # Se manda llamar la funcion
10 presentainfo( 10 )
11
12 presentainfo( 70, 60, 50 )
```

Otro ejemplo de una función con argumentos variables.

```
1  # Se define la funcion
2  def presentainfo( arg1, *vartupla ):
3      print ('La salida es: ')
4      print (arg1)
5      for var in vartupla:
6          print (var)
7      return
8
9  # Se manda llamar la funcion
10 presentainfo( 10 )
11
12 presentainfo( 70, 60, 50 )
```

Otro ejemplo de una función con argumentos variables.

```
1  import math
2
3  def atangente(*args ):
4  # args es una tupla de argumentos
5      if len(args) == 1:
6          return math.atan(args[0])
7      else :
8          for var in args:
9              print (var, ' - ', math.atan(var))
10     return
11
12 print (atangente (0.2))
13 print (atangente (2.0,10.0))
14 print (atangente (-2.0,-10.0, 3.14))
```

Funciones anónimas.

Estas funciones se llaman anónimas porque no se declaran de la manera estándar utilizando la palabra reservada `def`.

Se utiliza la palabra clave `lambda` para crear pequeñas funciones anónimas.

- Las funciones `lambda` pueden tomar cualquier número de argumentos pero devuelven sólo un valor. No pueden contener comandos ni expresiones múltiples.
- Una función anónima no puede ser una llamada directmente para imprimir porque requiere una expresión.
- Las funciones `lambda` tienen su propio espacio de nombres local y no pueden acceder a variables distintas de las de su lista de parámetros y las del espacio de nombres global.

Sintaxis de la función `lambda`.

```
1 # Aqui se define la funcion
2
3 suma = lambda arg1, arg2: arg1 + arg2
4
5
6 # Se llama a suma como una funcion
7 print ('Valor total: ', suma( 10, 20 ))
8 print ('Valor total: ', suma( 20, 20 ))
```

La instrucción `return`.

La instrucción `return [expresion]` sale de una función, opcionalmente devolviendo una expresión a la línea que llama a la función. Una sentencia `return` sin argumentos es igual que `return None`.

```
1  # Aqui se define la funcion
2  def suma2( arg1, arg2 ):
3      total = arg1 + arg2
4      print ('Dentro de la funcion : ', total)
5      return total
6
7  # Se llama a la funcion suma
8
9  resultado = suma2( 10, 20 )
10
11 print ('Fuera de la funcion : ', resultado )
```

Contenido

Alcance de las variables.

Es posible que todas las variables de un programa no estén accesibles para diferentes procesos de ese programa. Esto depende de dónde se haya declarado una variable.

El alcance de una variable determina la parte del programa donde se puede acceder a un identificador particular. Hay dos ámbitos básicos de variables en `python`:

- Variables globales.
- Variables locales.

Variables globales vs. locales.

Las variables que se definen dentro del cuerpo de una función tienen un *ámbito local* y las definidas en el exterior tienen un *ámbito global*.

Esto significa que las variables locales se pueden acceder sólo dentro de la función en la que se declaran, mientras que las variables globales se puede acceder a través del cuerpo del programa por todas las funciones.

Cuando se llama a una función, las variables declaradas dentro de ella se introducen en el ámbito.

Ejemplo del alcance de las variables.

```
1  total = 0 # Esta es una variable global
2
3  # Aqui se define la funcion
4  def suma3( arg1, arg2 ):
5      total = arg1 + arg2; # En este punto, total es
           una variable local.
6      print ('Dentro de la funcion la variable local,
           el total es : ', total)
7      return total
8
9  # Se llama a la funcion suma3
10 suma3( 10, 20 )
11 print ('Fuera de la funcion la variable global, el
       total es : ', total )
```

Contenido

Módulos en python.

Es una buena práctica almacenar las funciones definidas por el usuario en módulos.

Un módulo es un archivo en donde se dejan las funciones, el nombre del módulo es el nombre del archivo.

Archivo `.py` que define el módulo

```
1 def fib(n):
2     result = []
3     a, b = 0, 1
4     while b < n:
5         result.append(b)
6         a, b = b, a + b
7
8     return result
```

Archivo que manda llamar la función del módulo.

```
1 from fibonacci import fib
2
3 print(fib(100))
4
5 #salida en pantalla
6 [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```