Estructuras Discretas para la Computación

Repaso de Python

Tomás J. Concepción Miranda

Universidad Tecnológica de Panamá

Tabla de contenido

Historia de Python

- Creado por Guido van Rossum
- Influenciado por C, Perl, ALGOL, APL, Lisp, Ada, entre otros
- Lanzado en 1991 en su versión 0.9.0
- Desde 2023, su versión más reciente es la 3.11
- Uno de los lenguajes de programación más populares que existen

Un lenguaje de *script*

Lenguajes como la familia de C (C, C++, C#), Ada, Pacal, Java (en bytecode), Rust, Go, entre otros, son lenguajes compilados.

Lenguajes compilados

El programa fuente es transformado, usando un *compilador*, en lenguaje máquina para su ejecución.

Python, en cambio, es un lenguaje interpretado.

Lenguajes interpretados

El programa fuente es leído por un programa llamado *intérprete*, que ejecuta el programa fuente linea por linea.
A este programa fuente se le conoce como *script* (*guíon* en inglés, como el de un drama o una película).

El interprete de Python

- Permite ejecutar comandos de Python
- Tiene un historial

Ejecución de un programa Python

Se guarda el programa en un archivo *.py*, por ejemplo `hola.py`

- Mediante la linea de comandos: `python hola.py`
- Mediante ejecución en el IDE (muestra una terminal integrada)

Donde escribir programas Python

IDE (*integrated development environment*): aplicación para facilitar el desarrollo de programas.
Para Python existen:

- PyCharm
- VSCode
- ~~Notepad~~ Notepad++
- Varios otros (Vim, emacs)

Sintaxis

- Línea física: secuencia de caracteres terminados con un salto de línea
- Línea lógica: instrucción terminada con un salto de línea. Puede ser constituida de varias líneas físicas
- Comentarios
  - usando # el resto de la línea no se toma en cuenta
  - usado tres apostrofes ''' crea un comentario en bloque terminado en otros tres apostrofes (similar a `/* */` en C)
- Sangría (`indentation`): introducción de varios caracteres en blanco (espacios) al comenzar un párrafo (una línea en este caso). Todas las instrucciones dentro de bucles y las estructuras de control (`if`, `for`) se sangran

Tipos de datos: `int`, `float`, `complex`

- Dentro el intérprete se pueden realizar cálculos matemáticos con expresiones matemáticas
- suma, resta, multiplicación, división con los símbolos usuales
- `**` : potenciación
- `//` : división de enteros
- `%` : módulo

Variables

- Permiten guardar datos, usando un nombre como referencia
- Sus valores pueden cambiar a lo largo de la ejecución (de ahí su nombre)
- Python automáticamente asigna el tipo de dato a la variable, basta con declarar la variable. Es posible verificar el tipo de la variable con la función `type`

Tipos de datos: `str`

- Cadena de caracteres o *string*
- Se pueden construir con dos apostrofes: `'hello, world'`, o con dos comillas: `"hello, world"`
- La concatenación se realiza con el operador `+`
- Se pueden acceder a sus elementos con corchetes, *e.g.* `'hello, world'[7]` resulta en `w`

Tipos de datos: `list`

- Una colección con orden
- *mutable*: se puede modificar el valor de los elementos de las listas
- Puede contener valores de diferentes tipos
- Se declaran usando corchetes,
  e.g. `lista = [99, "hola", 1.4, 'a']`
- Se puede acceder a sus elementos usando corchetes al final de la lista, *e.g.* `lista[2]` resulta en `1.4`

Entrada: Lectura de datos desde la terminal

El ingreso de valores mediante la terminal se puede realizar con la función `input`. Ejemplo:

Listing 1: nombre.py

```
1 nombre = input("Ingrese su nombre: ")
2 print("Su nombre es " + nombre)
```

La función `input` siempre retorna una cadena. Si se quiere otro tipo de valor, se tiene que convertir el tipo cadena al tipo de dato que se quiere (si la conversión es posible).

Salida: cómo mostrar información

La función `print` permite imprimir una cadenas de caracteres en pantalla.
Dos métodos de formato:

- literales de cadenas de formato
- el método `str.format`

Literales de cadenas de formato

Se forman al usar `f` antes de empezar la cadena.

Listing 2: literales_cadenas.py

```
1 nombre = "Juan"
2 apellido = "Garcia"
3 print(f"Mi nombre es {nombre} {apellido}")
```

el código anterior produce:

```
Mi nombre es Juan Garcia
```

El método `str.format`

El método `format` esta integrado en las variables `str`:

Listing 3: str_format.py

```
1 num_arboles = 1300
2 prom_crecimiento = 1.4948
3 print("Los {:,} arboles crecen {:6.2f} cm por año".
      format(num_arboles, prom_crecimiento))
```

produce

```
Los 1,300 arboles crecen   1.49 cm por año
```

Manejan el orden en el que sentencias, instrucciones, o llamadas de funciones individuales son ejecutadas.
En Python tenemos:

- `while`
- `if`, `else`
- `break`
- `continue`
- `for`
- otras más...

Control de flujo - `while`

Repite una secuencia de instrucciones mientras se cumpla la condición inicial.

Listing 4: while.py

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

Control de flujo - `if`, `elif`, `else`

- `if`: ejecuta las instrucciones que contienen si la condición es verdadera
- `elif`: se coloca después de por lo menos un `if`, y se ejecuta si la sentencias `if` precedente no se ejecutó y se cumple su condición
- `else`: se coloca al final de por lo menos un `if`, y se ejecuta si ninguna de las sentencias `if` o `elif` precedentes se ejecutaron

Control de flujo - if, elif, else

Listing 5: if.py

```
1  x = int(input("Ingrese un entero: "))
2  if x < 0:
3      x = 0
4      print('Negativo cambiado a cero')
5  elif x == 0:
6      print('Cero')
7  elif x == 1:
8      print('Uno')
9  else:
10     print('Más')
```

Operadores lógicos

- Comparación (==, !=, >, <, >=, <=)
- and (y)
- or (o)
- not (negación)

Control de flujo - `for`

Repite una secuencia de instrucciones al iterar sobre los elementos de una secuencia (una lista o una cadena de caracteres) en el orden que aparecen en la secuencia.

Listing 6: for.py

```
1 palabras = ['gato', 'ventana', 'defenestrar']
2 for p in palabras:
3     print(p, len(p))
```

La función `range`

Permite crear una secuencia de números. Tiene varias opciones de argumentos que devuelven lo siguiente:

- `range(10)`: una secuencia del 0 al 9
- `range(0,10)`: igual al ejemplo anterior
- `range(0,10,2)`: una secuencia del 0 al 9 de dos en dos: 0, 2, 4, 6, 8

Esta función devuelve un valor de tipo `range`. Si se quiere la lista de elementos, se debe crear una lista usando: `list(range(10))`

Permite salir de un bucle en cualquier punto de su ejecución, es decir, permite terminar la ejecución el bucle. Si hay bucles anidados, `break` sale del bucle más anidado.

Listing 7: break.py

```
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//x)
5             break
6     else:
7         # el bucle terminó sin encontrar un factor
8         print(n, ' es un numero primo')
```

Control de flujo - continue

Permite recomenzar el bucle (con el siguiente valor si lo tiene) sin ejecutar el resto de instrucciones dentro del bucle.

Listing 8: continue.py

```
1  for num in range(2, 10):
2      if num % 2 == 0:
3          print("Número par ", num)
4          continue
5      print("Número impar ", num)
```

Declarar funciones

Las funciones se inicializan con la sentencia def, seguido de un nombre, y luego los argumentos entre paréntesis.

Listing 9: funcion.py

```
1  def fib(n):
2      a, b = 0, 1
3
4      while a < n:
5          print(a, end=' ')
6          a, b = b, a+b
7          print()
```

Más acerca de listas

Una colección *mutable* de elementos. Aquí hay unos cuantos métodos disponibles en las listas:

- `list.append(x)`: Agrega un valor al final de la lista
- `list.extend(iterable)`: Extiende la lista agregándole todos los valores del `iterable` (una lista, un `range`, entre otros objetos iterables)
- `list.insert(i, x)`: Inserta un valor en una posición dada. El primer argumento es el índice del valor delante del cual se insertará, por lo tanto `a.insert(0, x)` inserta al principio de la lista y `a.insert(len(a), x)` equivale a `a.append(x)`
- `list.remove(x)`: Quita el primer valor de la lista cuyo valor sea `x`. Lanza un `ValueError` si no existe tal valor

Comprensión de listas

Permite crear nuevas listas donde cada elemento es el resultado de operaciones realizadas a cada miembro de la secuencia.

Comprensión de listas

Listing 10: comprension_listas.py

```
1  vec = [-4, -2, 0, 2, 4]
2
3  # crear una lista nueva con valores doblados
4  print([x*2 for x in vec])
5
6  # filtrar la lista para excluir números negativos
7  print([x for x in vec if x >= 0])
8
9  # aplicar una función a todos los elementos
10 print([abs(x) for x in vec])
11
12 # llamar un metodo en cada elemento
13 frutasfrescas = ['  guineo', '  mango ', ' maracuyá ']
14 print([letra.strip() for letra in frutasfrescas])
```

Tuplas

- Una colección *inmutable* de elementos. Es decir, no se pueden cambiar los valores de la colección una vez haya sido creada
- Se pueden crear usando paréntesis o simplemente colocando los valores uno detrás del otro

Listing 11: tuplas.py

```
1  t = 12345, 54321, 'hello!'
2  print(t[0])
3  print(t)
4
5  # Las tuplas pueden ser anidadas:
6  u = t, (1, 2, 3, 4, 5)
7  print(u)
8
9  # Las tuplas son imutables:
10 t[0] = 88888
```

Diccionarios

Un diccionario permite asignar una *clave* (una cadena o un número) a un valor.
Se les puede considerar un conjunto *clave:valor*.

Listing 12: diccionarios.py

```
1 directorio_tel = {'juan' : 64835182, 'pedro' : 2309364}
2 print(directorio_tel['juan'])
```

El acceso a un valor del diccionario se realiza colocando una llave que existe en el diccionario, como en la línea 2 del Listing 12.

Módulos

Son archivos que contienen definiciones de constantes y funciones de Python. Permiten ser reutilizados en varios scripts sin tener que ser reescritos.

Listing 13: fibo.py

```
1  # Módulo números de Fibonacci
2
3  def fib(n):    # escribe la secuencia de Fibonacci hasta n
4      a, b = 0, 1
5      while a < n:
6          print(a, end=' ')
7          a, b = b, a+b
8      print()
9
10 def fib2(n):   # retorna la secuencia de Fibonacci hasta n
11     result = []
12     a, b = 0, 1
13     while a < n:
14         result.append(a)
15         a, b = b, a+b
16     return result
```

La sentencia `import`

Permite cargar módulos dentro de una ejecución de un script Python. Es similar a la sentencia `include` en C.
Ejemplo con el archivo anterior `fibo.py`:

Listing 14: import.py

```
1 import fibo
2 fibo.fib(4)
```

Otra manera es de solo importar las funciones que nos interesan usando `from <modulo> import <funcion>, <funcion>,...`:

Listing 15: from_import.py

```
1 from fibo import fib, fib2
2 fib(500)
```

La sentencia `import`

Para importar todas las funciones, basta con poner un `*` después de `from ... import`:

Listing 16: import_asterisco.py

```
1 from fibo import *
2 fib(500)
```

Es posible "nombrar" un módulo usando la palabra `as`:

Listing 17: import_as.py

```
1 import fibo as fib
2 fib.fib(500)
```

En este caso, se renombra el módulo `fibo` en `fib`.

Instalar módulos

Muchos módulos externos se pueden instalar mediante el uso del programa `pip`: `pip install <modulo>`

Errores de sintaxis

Diferentes errores se pueden producir durante la ejecución. El más común de ellos serían los errores de sintaxis:

```
1  >>> while True print('Hello world')
2    File "<stdin>", line 1
3      while True print('Hello world')
4                       ^
5  SyntaxError: invalid syntax
```

Excepciones

Errores que el interprete detecta al ejecutar una instrucción. Instrucciones sintácticamente correctas pueden producir estos errores.

Excepciones

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Para "atrapar" estos errores, se coloca la instrucción después de un `try`. Un `except` y el tipo de excepción se coloca después del `try`, y ejecuta las instrucciones que contiene si se produce el tipo de excepción declarado.

Listing 18: gestion_excepciones.py

```
1 while True:
2     try:
3         x = int(input("Por favor ingrese un número: "))
4         break
5     except ValueError:
6         print("Oops!  Ese no es un número valido.
              Intente de nuevo...")
```

Clases

Una *clase* es una definición de un tipo de *objeto*, que puede contener datos y funcionalidades. Cada variable en Python es un objeto, una *instancia* de una clase.
Ya hemos trabajado con algunas de las clases integradas en Python:

- `str`
- `int`
- `float`
- `list`
- `range`
- `dict`

Librería estándar

Ofrece módulos incorporados (escritos en C) que brindan acceso a las funcionalidades del sistema como entrada y salida de archivos que serían de otra forma inaccesibles para los programadores en Python, así como módulos escritos en Python que proveen soluciones estandarizadas para los diversos problemas comunes.

Módulo sys

Permite, por ejemplo, obtener los argumentos de cuando se llamó el script. Estos se encuentran en una lista llamada `argv` de este módulo.

Listing 19: modulo_sys.py

```
1 import sys
2 print(sys.argv)
```

Módulo math

El módulo `math` permite el acceso a las funciones de la biblioteca C subyacente para realizar operaciones matemáticas de punto flotante:

Listing 20: modulo_math.py

```
1 import math
2 print(math.cos(math.pi / 4))
3 print(math.log(1024, 2))
```

Módulo statistics

El módulo statistics calcula propiedades de estadística básica (la media, mediana, varianza, etc) de datos numéricos:

Listing 21: modulo_statistics.py

```
1 import statistics
2 data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
3 print(statistics.mean(data))
4 print(statistics.median(data))
5 print(statistics.variance(data))
```

Librerías populares

- Librerías hechas y mantenidas por individuos y comunidades
- Proveen una gran cantidad de funcionalidades listas para usar
- Cada librería tiene propósitos diferentes de uso

Provee un objeto de arreglo multidimensional, varios objetos derivados, y una serie de rutinas para operaciones rápidas en arreglos.

Listing 22: ejemplo_numpy.py

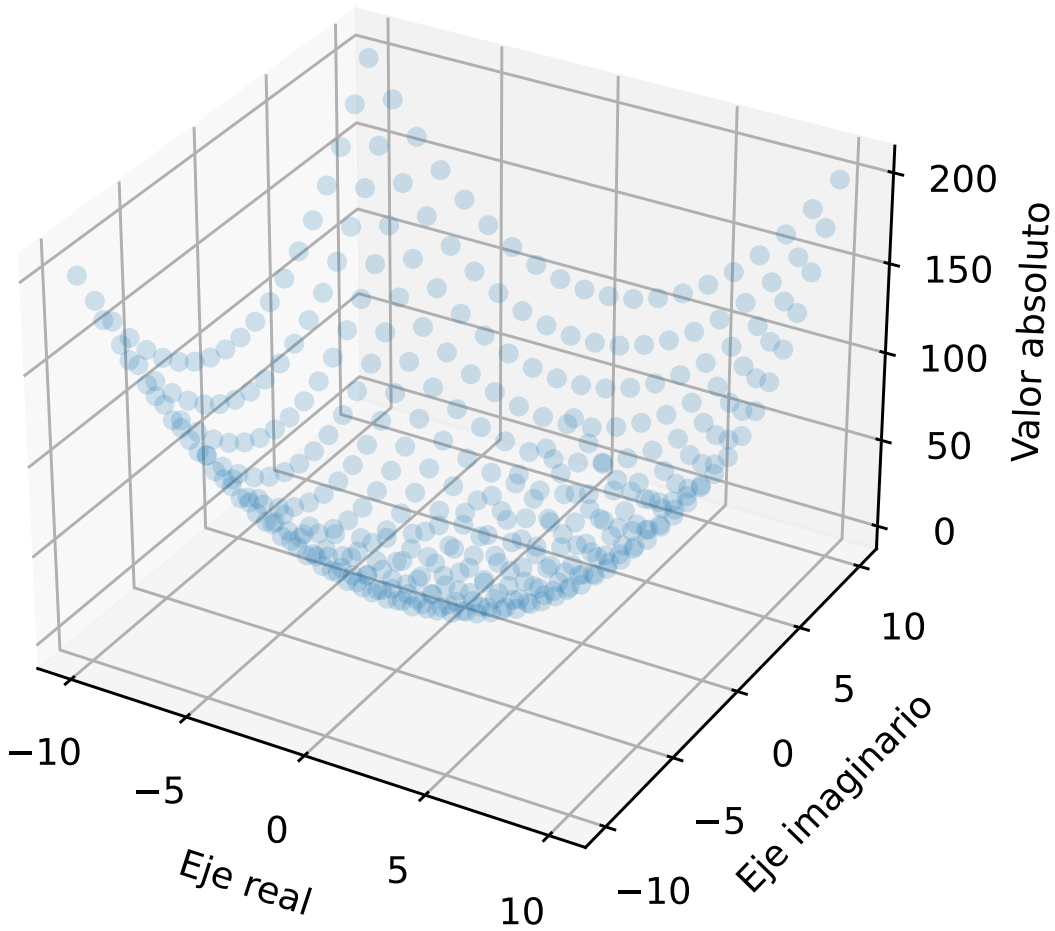
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.axes_grid1 import make_axes_locatable
4
5  def f(z):
6      return np.square(z) - 1
7
8  z = [4, 1-0.2j, 1.6]
9  print(f(z))
10
11 x, y = np.meshgrid(np.linspace(-10, 10, 20), np.linspace
       (-10, 10, 20))
12 mesh = x + (1j * y)  # Crear una malla de plano complejo
13
14 output = np.abs(f(mesh))  # Tomar el valor absoluto de la
       salida (para graficar)
```

Provee funciones para crear visualizaciones estáticas, animadas, e interactivas en Python.

Listing 23: ejemplo_numpy.py

```
1
2  fig = plt.figure()
3  ax = plt.axes(projection='3d')
4
5  ax.scatter(x, y, output, alpha=0.2)
6
7  ax.set_xlabel('Eje real')
8  ax.set_ylabel('Eje imaginario')
9  ax.set_zlabel('Valor absoluto')
10 ax.set_title('Una Iteracion: $ f(z) = z^2 - 1$');
11
12 plt.savefig("plot.pdf", format="pdf")
13 plt.show()
```

Una Iteracion: $f(z) = z^2 - 1$



Provee funciones para análisis y manipulación de datos.

Listing 24: ejemplo_pandas.py

```
1  import pandas
2  df = pandas.DataFrame(
3      {
4          "Name": [
5              "Braund, Mr. Owen Harris",
6              "Allen, Mr. William Henry",
7              "Bonnell, Miss. Elizabeth",
8          ],
9          "Age": [22, 35, 58],
10         "Sex": ["male", "male", "female"],
11     }
12 )
13
14 print(df)
```

Provee módulos, juego de datos, y tutoriales para el desarrollo de procesamiento de lenguajes naturales (PLN, en inglés *natual language processing* o NLP)

Listing 25: ejemplo_nltk.py

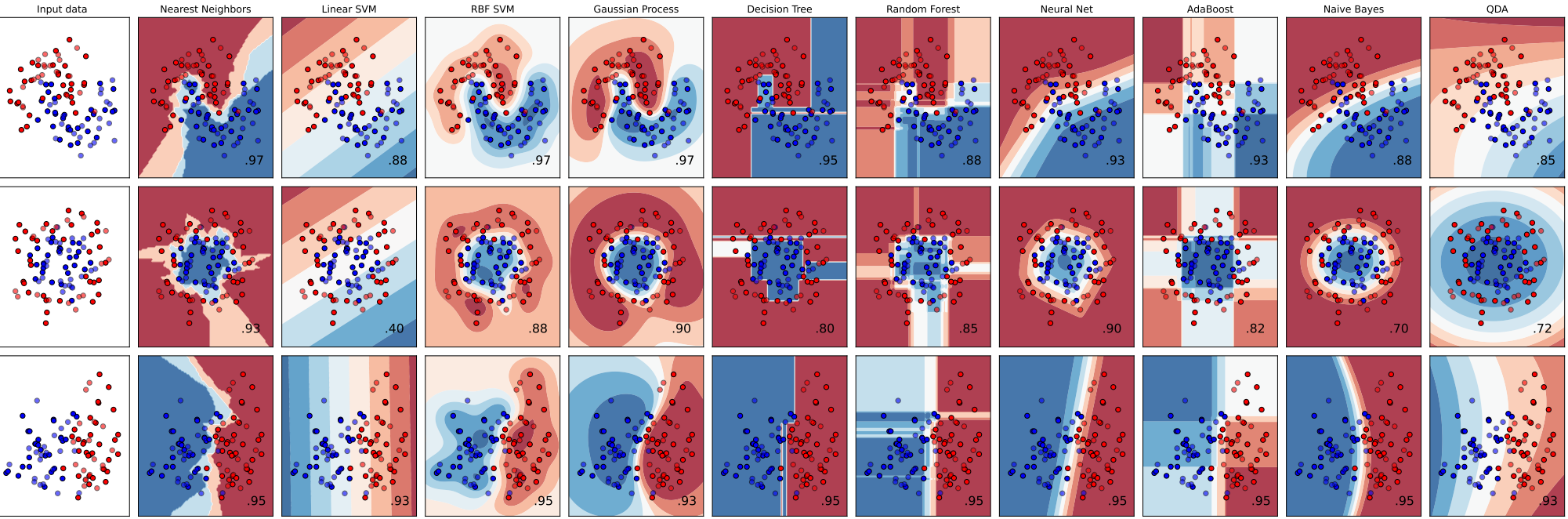
```
1  import nltk
2  nltk.download('punkt')
3  nltk.download('averaged_perceptron_tagger')
4  sentence = """At eight o'clock on Thursday morning...
       Arthur didn't feel very good."""
5  tokens = nltk.word_tokenize(sentence)
6  print(tokens)
7  tagged = nltk.pos_tag(tokens)
8  print(tagged[0:6])
9  entities = nltk.chunk.ne_chunk(tagged)
10 print(entities)
```

Provee funciones para hacer estadística de datos, análisis de datos, y algunos algoritmos de inteligencia artificial.

Listing 26: plot_classifier_comparison.py

```
59  classifiers = [
60      KNeighborsClassifier(3),
61      SVC(kernel="linear", C=0.025),
62      SVC(gamma=2, C=1),
63      GaussianProcessClassifier(1.0 * RBF(1.0)),
64      DecisionTreeClassifier(max_depth=5),
65      RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
66      MLPClassifier(alpha=1, max_iter=1000),
67      AdaBoostClassifier(),
68      GaussianNB(),
69      QuadraticDiscriminantAnalysis(),
70  ]
71
72  X, y = make_classification(
73      n_features=2, n_redundant=0, n_informative=2, random_state=1, n_clusters_per_class=1
74  )
75  rng = np.random.RandomState(2)
76  X += 2 * rng.uniform(size=X.shape)
77  linearly_separable = (X, y)
78
79  datasets = [
80      make_moons(noise=0.3, random_state=0),
81      make_circles(noise=0.2, factor=0.5, random_state=1),
82      linearly_separable,
83  ]
```

scikit-learn



Referencias

- Tutorial de la doc oficial de Python (https://docs.python.org/es/3/tutorial/index.html)
- Referencia del lenguaje (https://docs.python.org/es/3/reference/index.html)