Neural Networks: Python control flow tools and basic libraries

Andrzej Kordecki

Neural Networks for Classification and Identification (ML.EM05): Exercise 03-04
Division of Theory of Machines and Robots
Institute of Aeronautics and Applied Mechanics
Faculty of Power and Aeronautical Engineering
Warsaw University of Technology

Table of Contents

Python Control Flow Tools

Control Flow Tools

The Python interpreter executes code in a line-by-line manner. But, we can change it with "control flow" tools in your code. These tools will affect the order in which the code in your program is executed. Example of most commonly used flow tools:

- if
- for
- while

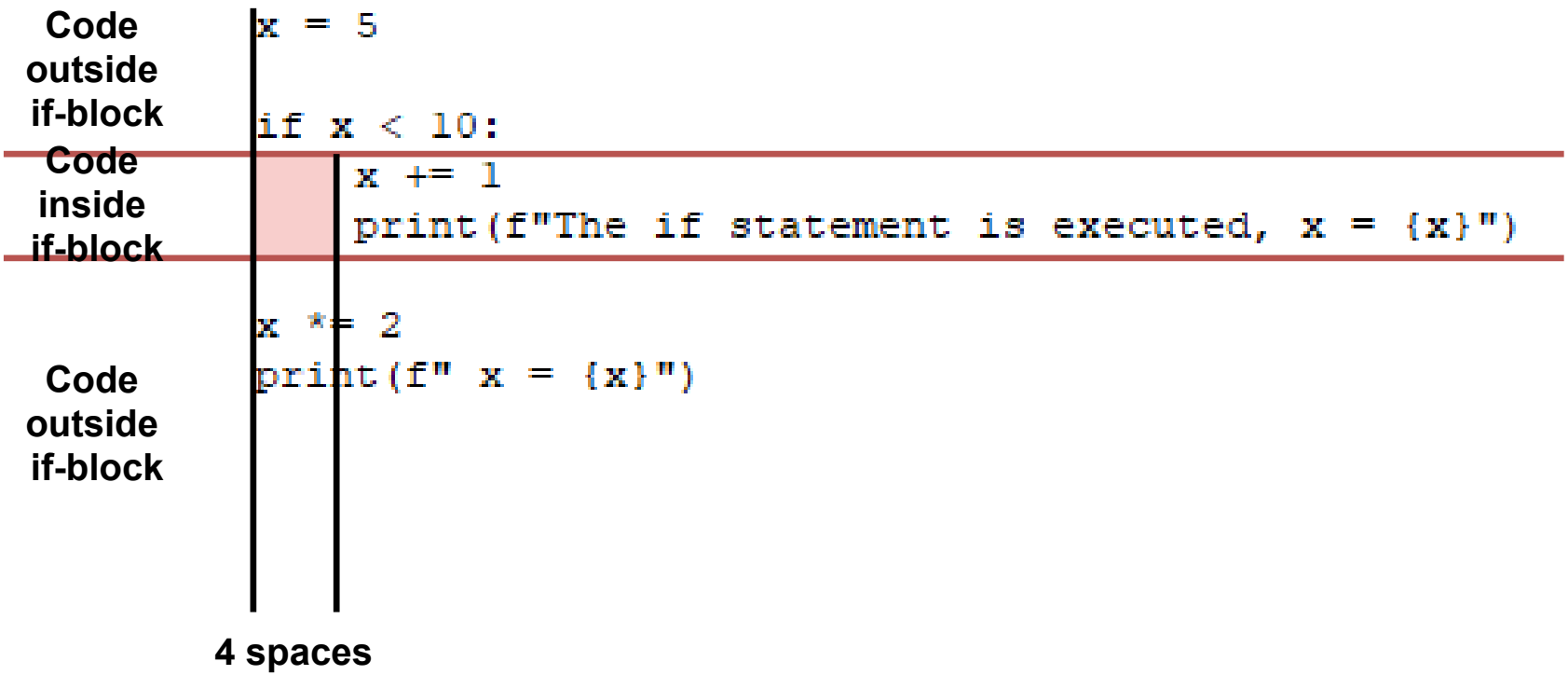
There are many others control functions e.g.: else, continue, break, match (3.10).

Importance of whitespace

The concepts of function definitions, loops, and conditional statements are sharing similar syntax.

Python uses **whitespace** (4 spaces) to delimit scope

Example of "if" statement:



Coding Style PEP 8

Most languages can be written in different styles, to make it easier for others to read your code. For Python, PEP 8 has emerged as the style guide that most projects adhere to. It promotes a very readable and eye-pleasing coding style.

- Use 4-space indentation, and no tabs. The 4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don't exceed 79 characters. This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

Coding Style PEP 8

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own. Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1, 2) + g(3, 4).
- Name your classes and functions consistently; the convention is to use UpperCamelCase for classes and lowercase_with_underscores for functions and methods. Always use self as the name for the first method argument.

Coding Style PEP 8

- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

More information on webpage:
`https://www.python.org/dev/peps/pep-0008/`

if Statements

What is the result of the example?

```
x = 5
if x < 10:
    x += 1
    print(f"The if statement is executed, x = {x}")
x *= 2
print(f"x = {x}")

# Prints
# The if statement is executed, x = 6
# x = 12
```

if Statements

The results will change, if we increase x value at the beginning.

```
x = 11
if x < 10:
    # the lines inside if statement were not executed
    x += 1
    print(f"The if statement is executed, x = {x}")
x *= 2
print(f"x = {x}")

# Prints
# x = 22
```

if Statements

The Python if-elif-else statement.

```
# The input() function allows user
# input data with keyboard
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

if Statements

There can be zero or more elif parts, and the else/elif part is optional.

```
# For input x = 40 # Print: "More"
# For input x = 1  # Print: "Single"
# For input x = 0  # Print: "Zero"
# For input x = -1 # Print: "Negative changed to zero"
```

for loop statement

The "for" statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

```
words = ['cat', 'window', 'truck']
for w in words:
    print(f"word w = {w}, length = {len(w)}")

# Prints:
# word w = cat, length = 3
# word w = window, length = 6
# word w = truck, length = 5
```

for Loop statement

If you do need to iterate over a sequence of numbers, you can use the built-in function range().

```
# Sequence of numbers from 0 to 5
x = range(6)
for n in x:
    print(n)

# Prints:
# 0 # 1 # 2 # 3 # 4 # 5
```

The command range(0, 10, 3) will create sequence: 0, 3, 6, 9.

for loop statement

Instead of putting the iterable directly after in in the for loop, you put enumerate() inside the loop.

```
i = 0
for w in words:
    i += 1
    print(f"no={i}, word w={w}, length={len(w)}")

for j, w in enumerate(words):
    print(f"no={j}, word w={w}, length={len(w)}")

# Both prints:
# no=1, word w=cat, length=3
# no=2, word w=window, length=6
# no=3, word w=truck, length=5
```

while Loop statement

With the while loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
    print(i)
    i += 1

# Prints:
# 1
# 2
# 3
# 4
# 5
```

break and continue Statements

Control over loops:

- The break statement breaks out of the innermost enclosing "for" or "while" loop.
- The "continue" statement continues with the next iteration of the loop:

```
n = 0
for num in range(1, 2):
    if num % 2 == 0:
        n += 1
        print("Found an even number", num)
        continue
    print("Found an odd number", num)
# Found an odd number 1
# Found an even number 2
```

break and continue Statements

```
n = 0
for num in range(1, 20):
    if num % 2 == 0:
        n += 1
        print("Found an even number", num)
        continue
    if n == 2:
        print(f"We have found {n} even numbers")
        break
    print("Found an odd number", num)
# Found an odd number 1
...
# Found an even number 4
# We have found 2 even numbers
```

Function def

A function is a block of code which only runs when it is called. We can pass data (known as parameters) into a function and return data as a result. The keyword def introduces a function definition.

```
# Function definition
def my_function():
    print("Inside of function")

# Calling a Function
my_function()

# Prints result:
# Inside of function
```

Function def

Information can be passed into functions as arguments.

```
def my_function(name):
  print(f"Name: {name}")

my_function("Bill")
my_function("Tom")

# Prints:
# Name: Bill
# Name: Tom
```

Function def

We can pass many function arguments of different types with default values.

```
def my_function(number, name = "Andrzej"):
  print(f"Number: {number}, Name: {name}")

my_function(5)

# Prints:
# Number: 5, Name: Andrzej
```

Function def

If you do not know how many arguments that will be passed into your function, add a "*" (asterisk) before the parameter name in the function definition.

```
def my_function(*names):
    for x in names:
        print("Me colleague name: " + x)

my_function("Bill", "Tom", "Harry")

# Prints:
# Me colleague name: Bill
# Me colleague name: Tom
# Me colleague name: Harry
```

Function def

Functions can also be called using keyword arguments of the form key = value syntax.

```
def my_function(name1, name2, name3):
    print("Me colleague name1: " + name1)
    print("Me colleague name2: " + name2)
    print("Me colleague name3: " + name3)

my_function(name1="Bill", name2="Tom", name3="Harry")
# or my_function("Bill", name2="Tom", name3="Harry")

# Prints:
# Me colleague name1: Bill
# Me colleague name2: Tom
# Me colleague name3: Harry
```

Lambda Expressions

Small anonymous functions can be created with the lambda keyword. They are syntactically restricted to a single expression.

```
count_a = lambda a: a + 113

def count_b(a):
    return a + 113

print(count_a(5))
print(count_b(5))

# Prints:
# 118
# 118
```

Lambda Expressions

Lambda functions can use many arguments and can be used inside nested function.

```
y = lambda a, b, c = 1: a + b - c
print(y(4, 6, 7)) # Print "y = 3"

def count_c(n, m):
    return lambda a: a*n+m

d1 = count_c(5, 5)
print(f"d1 = {d1(2)}") # Print "d1 = 15"

d2 = count_c(5, 5)(2)
print(f"d2 = {d2}") # Print "d2 = 15"
```

Object-oriented programming

Object-oriented programming (OOP) combines a group of data attributes with functions or methods into a unit called an "object." Fundamental concepts of OOP:

- Inheritance,
- Encapsulation,
- Abstraction,
- Polymorphism.

Creating a new class creates a new type of object.

Classes

The simple form of class definition with instantiation:

```
class MyClass:
    def myinfo(self, id):
        print(f"Hello {id}")

# create a new object/instance of a class.
k1 = MyClass()
print(k1)
# Print <__main__.MyClass object at 0x7f7115f45710>

k1.myinfo(89) # Prints "Hello 89"
```

Classes

The class constructor assign the values to the data members of the class when an object of the class is created.

```
class MyClass:
    color = "red" #attribute shared by all instances

    def __init__(self, id):
        self.id = id # new instance attribute

    def myinfo(self):
        print(f"Hello {self.id}")

k1 = MyClass(89)
k1.myinfo() # Prints "Hello 89"
print(k1.id) # Prints "89"
```

Inheritance

Inheritance is the procedure in which one class inherits the attributes and methods of another class. Classes can be organized into hierarchies, where a class might have one or more parent or child classes.

- The class whose properties and methods are inherited is known as Parent class.
- The class that inherits the properties from the parent class is the Child class.

The child class can have its own properties and methods.

Inheritance

Inheritance syntax:

```
class parent_class:
body of parent class

class child_class( parent_class):
body of child class
```

Inheritance

Inheritance example:

```
class Car:  #parent class
    def __init__(self, name, model):
        self.name = name
        self.model = model

    def informations(self):
        return f"Car brand {self.name}, \
            model {self.model}"
```

Inheritance

```
class Opel(Car):      #child class
    pass

class Audi(Car):      #child class
    def audi_info(self):
        return "Audi class description."

car1 = Opel("Opel","Corsa")
print(car1.informations())
% Print "Car brand Opel, model Corsa"

car2 = Audi("Audi","A8")
print(car2.informations()) % Car brand Audi, model A8
print(car2.audi_info()) % Audi class description.
```

Encapsulation

The word, "encapsulate," means to enclose something.

- We encapsulate by binding the data and functions which operate on that data into a single unit, the class.
- We can hide private details of a class from the outside world and only expose functionality that is important for interfacing with it.
- When a class does not allow calling code access to its private data directly, we say that it is well encapsulated.

Encapsulation

You can declare the methods or the attributes:

- Protected by using a single underscore "_" before their names, e.g. self._name or def _method()
- Private by using a double underscore "__" before their names, e.g. self.__name or def __method()

You can access protected variables and methods which outside class, but can inherit them. But, you cannot access private variables and methods from outside the class.

Encapsulation

Encapsulation example:

```
class Car:  #parent class
    def __init__(self, name, model):
        self._name = name  # Protected
        self.__model = model  # Private

    def informations(self):
        return f"Car brand {self.name}, \
            model {self.model}"

car1 = Car("Opel", "Corsa")
print(car1._name)  % Print "Opel"
print(car1.__model) % Print error
```

Polymorphism

The polymorphism means having many forms (in Greek "poly"-many and "morph"-forms). It refers to the functions having the same names but carrying different functionalities. Polymorphism allows for the uniform treatment of classes in a hierarchy. Example:

```
class Car:  #parent class
  def description(self):
        return f"Car brand {self.name}, \
            model {self.model}"
```

Polymorphism

```
class Opel(Car):
  def description(self):
    print("Opel class informations.")

class Audi(Car):
  def description(self):
    print("Audi class informations.")

ocar, acar = Opel(), Audi()
for car in (ocar, acar):
    car.description()
# Print:
# Opel class informations.
# Audi class informations.
```

Abstraction

We use Abstraction for hiding the internal details or implementations of a function and showing its functionalities only.

- An abstract method is a method that is declared, but contains no implementation.
- Any class with at least one abstract function is an abstract class.
- We cannot create an object for the abstract class with the abstract method.

Abstraction

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abc.abstractmethod
    def area(self):
        pass
class Rectangle(Shape):
    def __init__(self, x,y):
        self.l = x
        self.b=y
    def area(self):
        return self.l*self.b

r = Rectangle(10,20)
print ('area: ',r.area()) # Print "200"
```

Numpy

Numpy

Numpy is fundamental package for scientific computing. It contains among other things:

- efficient calculation with N-dimensional array objects,
- efficient multi-dimensional container of generic data,
- broadcasting functions,

Importing library into Python code:

```
import numpy as np
```

Numpy

A numpy array is a grid of values, all of the same type, and is indexed by nonnegative integers:

```
a = np.array([1, 2, 3])
print(type(a))  # Prints "<class 'numpy.ndarray'>"
print(a.shape)  # Prints "(3,)"

# Change an element of the array
print(a[0], a[1], a[2])  # Prints "1 2 3"
a[0] = 0
print(a)                 # Prints "[0 2 3]"

b = np.array([[1,2,3],[4,5,6]])
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 1]) # Prints "1 2 5"
```

Numpy

Functions to create predefined arrays:

```
# Create an array of all zeros
a = np.zeros((2,2))
print(a)                  # Prints "[[ 0.  0.]
                          #          [ 0.  0.]]"
# Create an array of all ones
b = np.ones((2,2))
print(b)                  # Prints "[[ 1.  1.]]"
                          #          [ 1.  1.]]"
# Create a constant array
c = np.full((2,2), 9)
print(c)                  # Prints "[[ 9.  9.]
                          #          [ 9.  9.]]"
```

Numpy

Functions to create predefined arrays:

```
# Create a 2x2 identity matrix
d = np.eye(2)
print(d)                 # Prints "[[ 1.  0.]
                         #          [ 0.  1.]]"

# Create an array filled with random values
e = np.random.random((2,2))
print(e)
# Prints "[[ 0.87230167  0.81443941]
#     [ 0.23624134  0.87236687]]"
```

Numpy

Numpy arrays can be sliced:

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3]
print(b)          # Prints "[[2 3],
                  #          [6 7]]

# b[0, 1] is a reference to a[0, 2]
print(b[0, 1])  # Prints "3"
b[0, 0] = 9
print(a[0, 1])  # Prints "9"
```

Numpy

Boolean array indexing:

```
a = np.array([[1,2], [3, 4], [5, 6]])

# Elements of a that are bigger than 2;
bool_idx = (a > 2)
print(bool_idx)       # Prints "[[False False  True]
                      #          [ True  True  True]]

# We can use boolean array for indexing
print(a[bool_idx])  # Prints "[3 4 5 6]"
print(a[a > 2])     # Prints "[3 4 5 6]"
# We can use tuple for indexing
idx = ((0, 1), (1,2))
print(a[idx])       # Prints "[2 6]"
```

Numpy

Basic mathematical functions operate elementwise on arrays:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum
print(x + y)
print(np.add(x, y))
# Result [[ 6.0  8.0]
#     [10.0 12.0]]
```

Numpy

Note that unlike MATLAB, $*$ is elementwise multiplication, not matrix multiplication.

```
# Elementwise product
print(x * y)
print(np.multiply(x, y))
# Result [[ 5.0 12.0]
#      [21.0 32.0]]

# Elementwise division
print(x / y)
print(np.divide(x, y))
# Result [[ 0.2         0.33333333]
#      [ 0.42857143  0.5         ]]
```

Numpy

We can reshape or transpose a matrix

```
x = np.array([[0, 1], [2, 3], [4, 5]])
print(x)
# Prints [[0, 1],
          [2, 3],
          [4, 5]])

y = np.reshape(a, (2, 3)) # C-like index ordering
print(y)
# Prints [[0, 1, 2],
          [3, 4, 5]])
```

Numpy

Broadcasting is a mechanism that allows to work with arrays of different shapes when performing arithmetic operations.

```
x = np.array([[1,2,3],[4,5,6],[7,8,9],[10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)

# Add the vector v to each row of the matrix x
for i in range(4):
    y[i, :] = x[i, :] + v
print(y)
# Print [[ 2  2  4] [ 5  5  7] [ 8  8 10] [11 11 13]]
```

Matplotlib

Numpy

Matplotlib is Python data visualization library.

- supports a very wide variety of graphs and plots: histogram, bar charts, power spectra, error charts etc.
- the module named pyplot allows to control features of the plot: line styles, font properties, formatting axes, etc.
- Matplotlib is used along with NumPy.

Importing library into Python code:

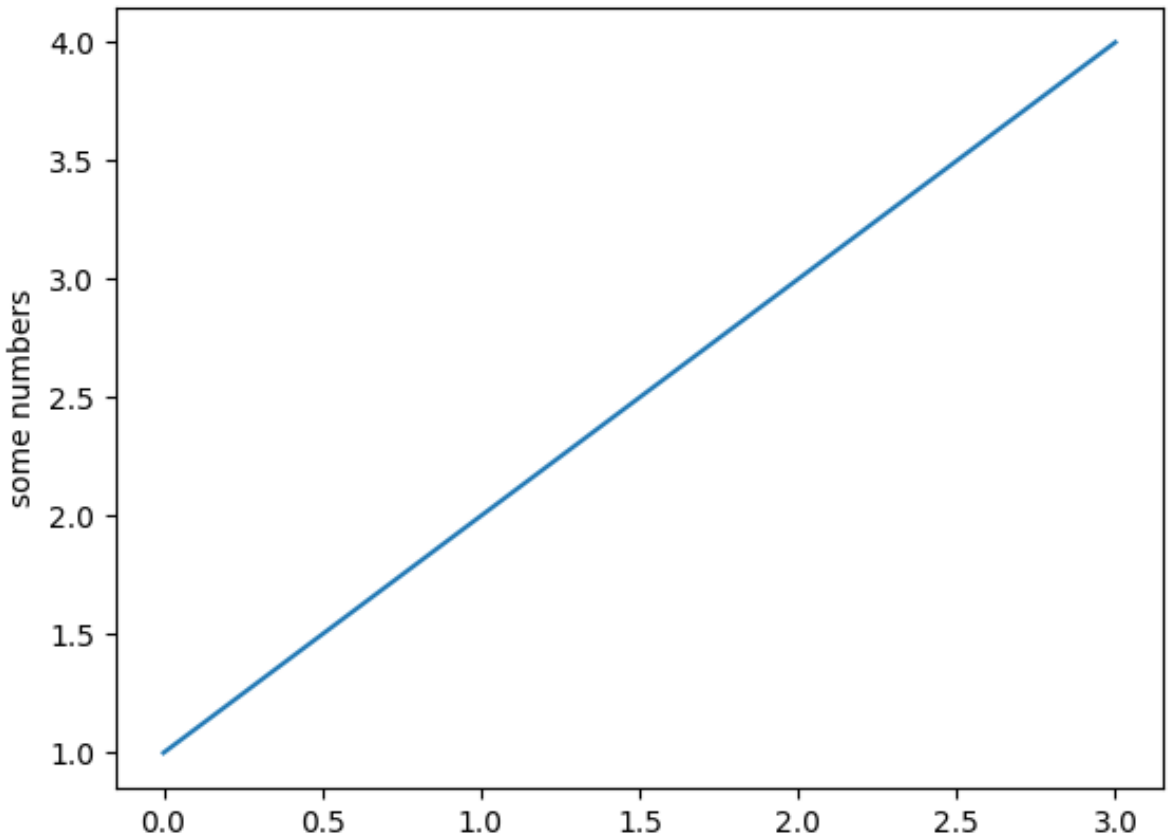
```
import numpy as np
import matplotlib.pyplot as plt
```

Plotting

Pyplot in Matplotlib is a collection of command style functions that make matplotlib work like MATLAB in showing data. The most important function in matplotlib is plot, which allows you to plot 2D data.

```
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```

Plotting

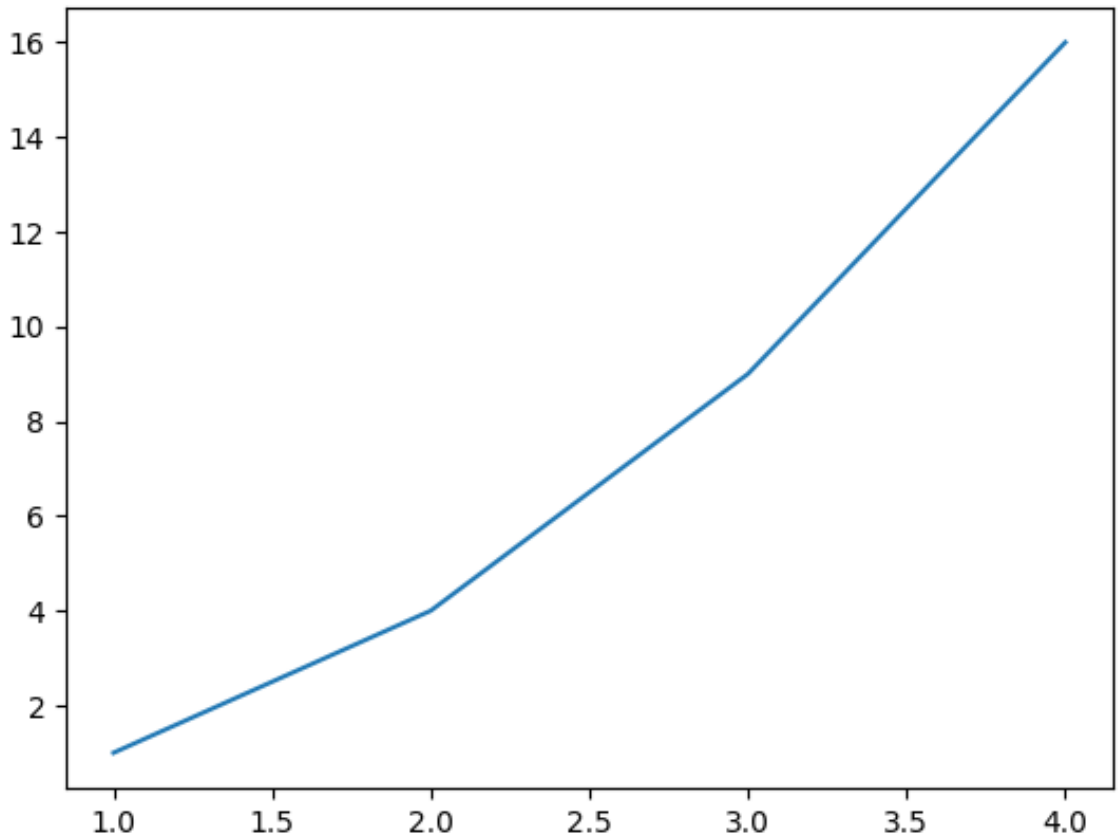


Plotting

Function plot() can take an arbitrary number of arguments, e.g. plot x versus y:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.ylabel('some numbers')
plt.show()
```

Plotting

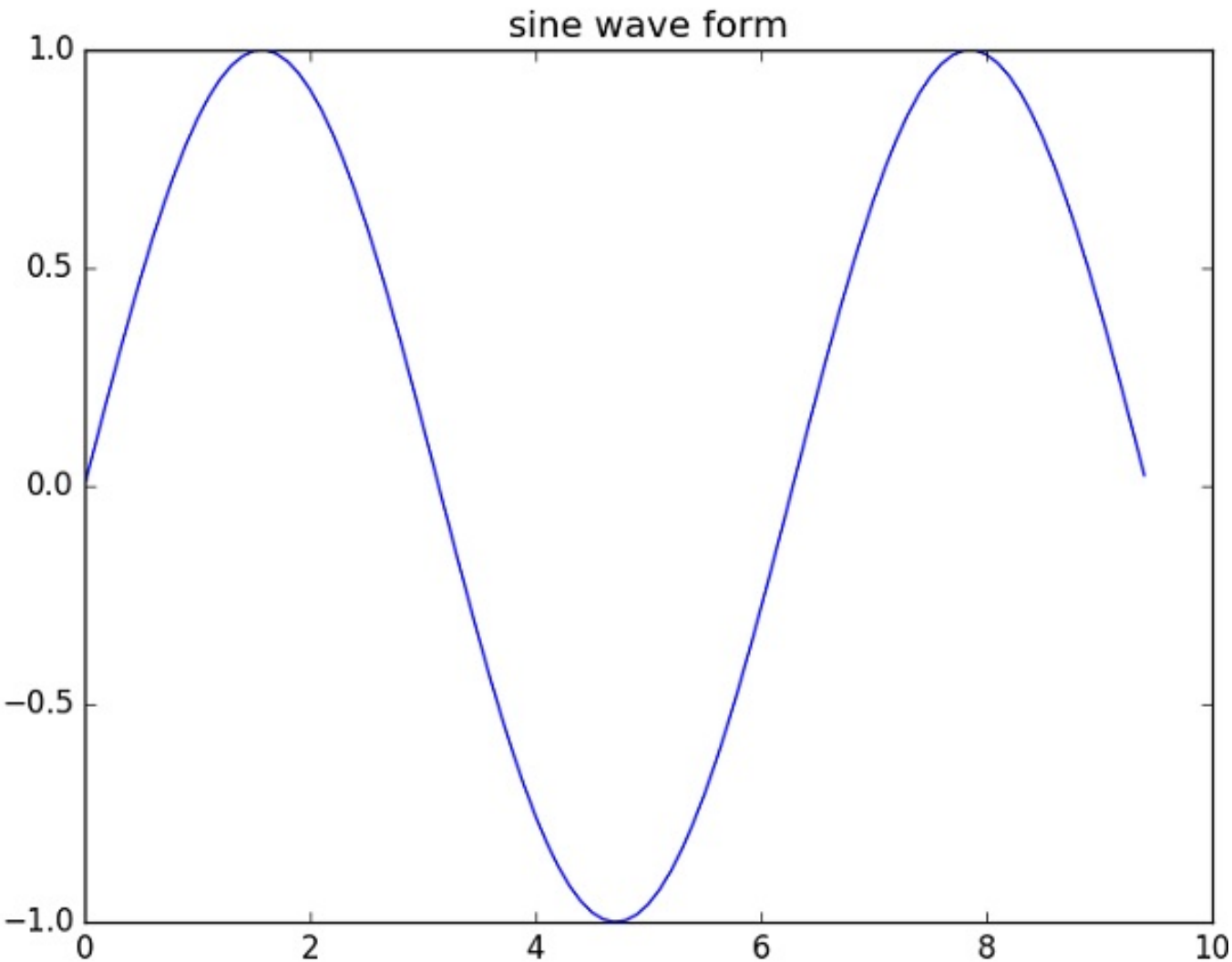


Plotting

Pyplot function allows to apply some changes to a figure: e.g., creates a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

```
# Compute the x and y coordinates for points on
# a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
plt.title("sine wave form")

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```

Plotting

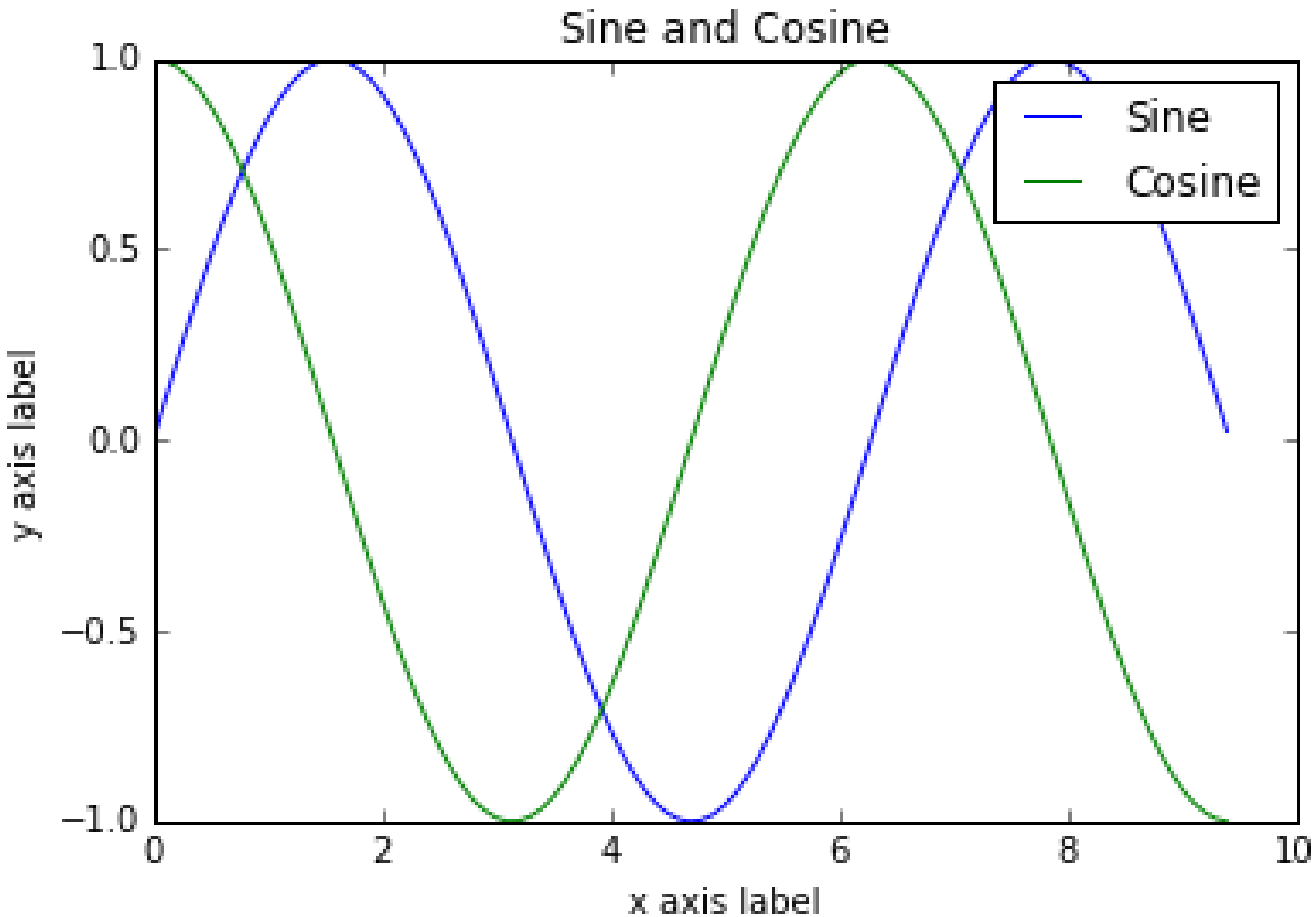


Plotting

We can easily plot multiple lines at once, and add a title, legend, axis labels:

```
# Compute the x and y coordinates for points on
# sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin), plt.plot(x, y_cos)
plt.xlabel('x axis label'), plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

Plotting

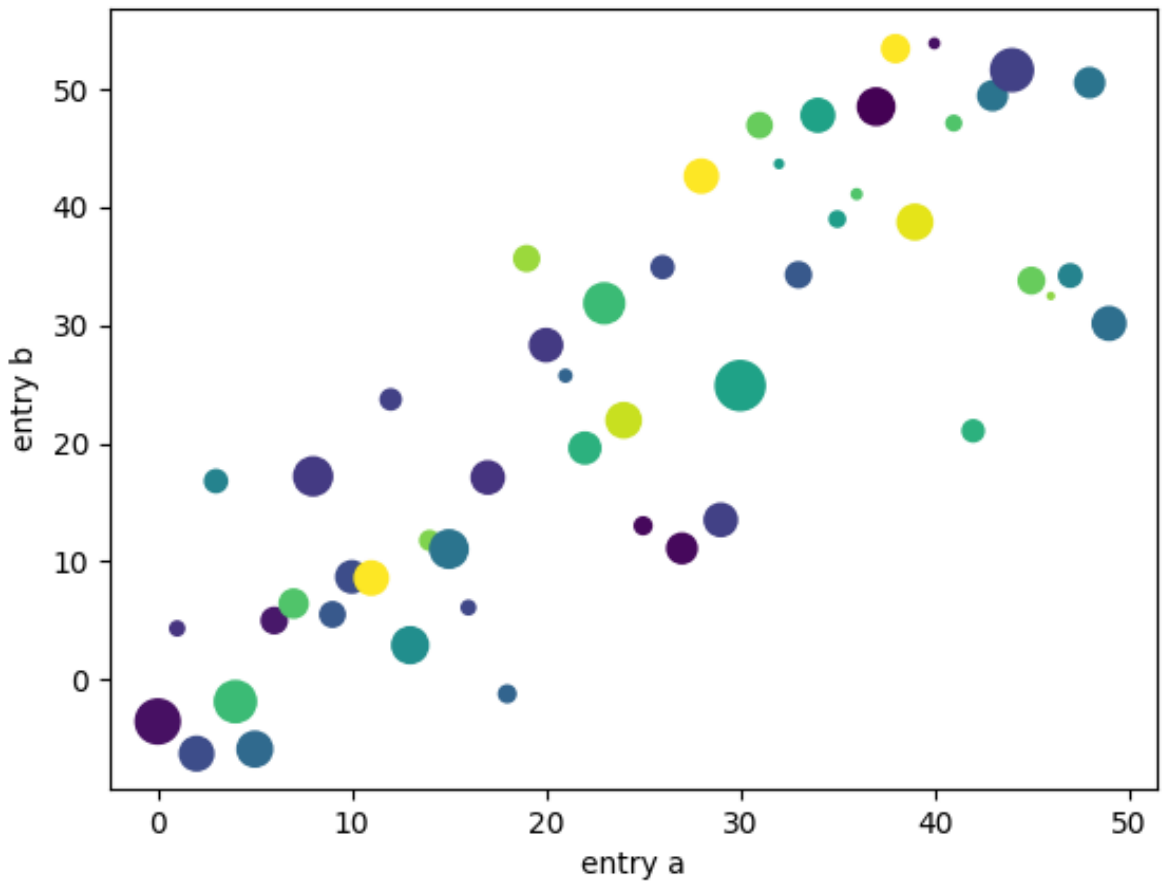

Sine and Cosine
Sine
Cosine
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10
y axis label
x axis label

Plotting

Matplotlib may generate plots with the strings corresponding to object with the data keyword argument.

```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```

Plotting

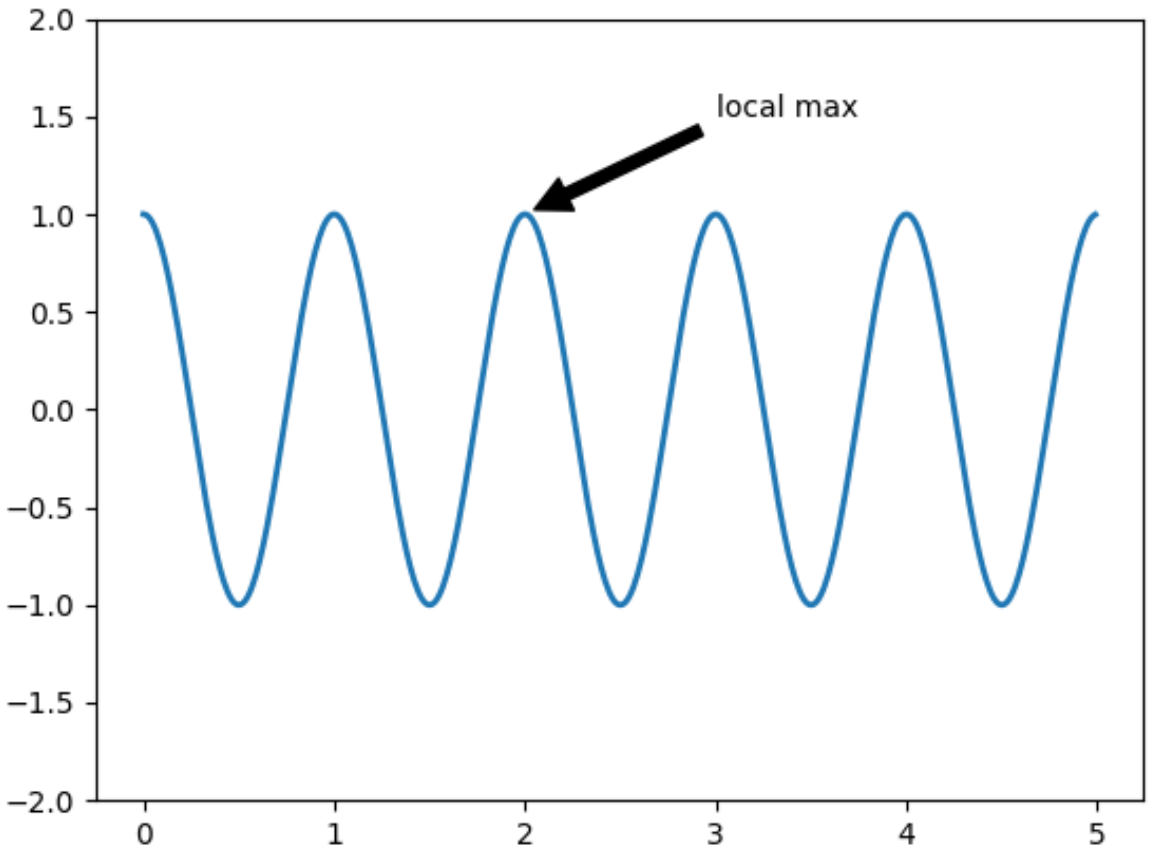


Plotting

Annotating text.

```
ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black',
                             shrink=0.05))

plt.ylim(-2, 2)
plt.show()
```

Plotting



Subplots

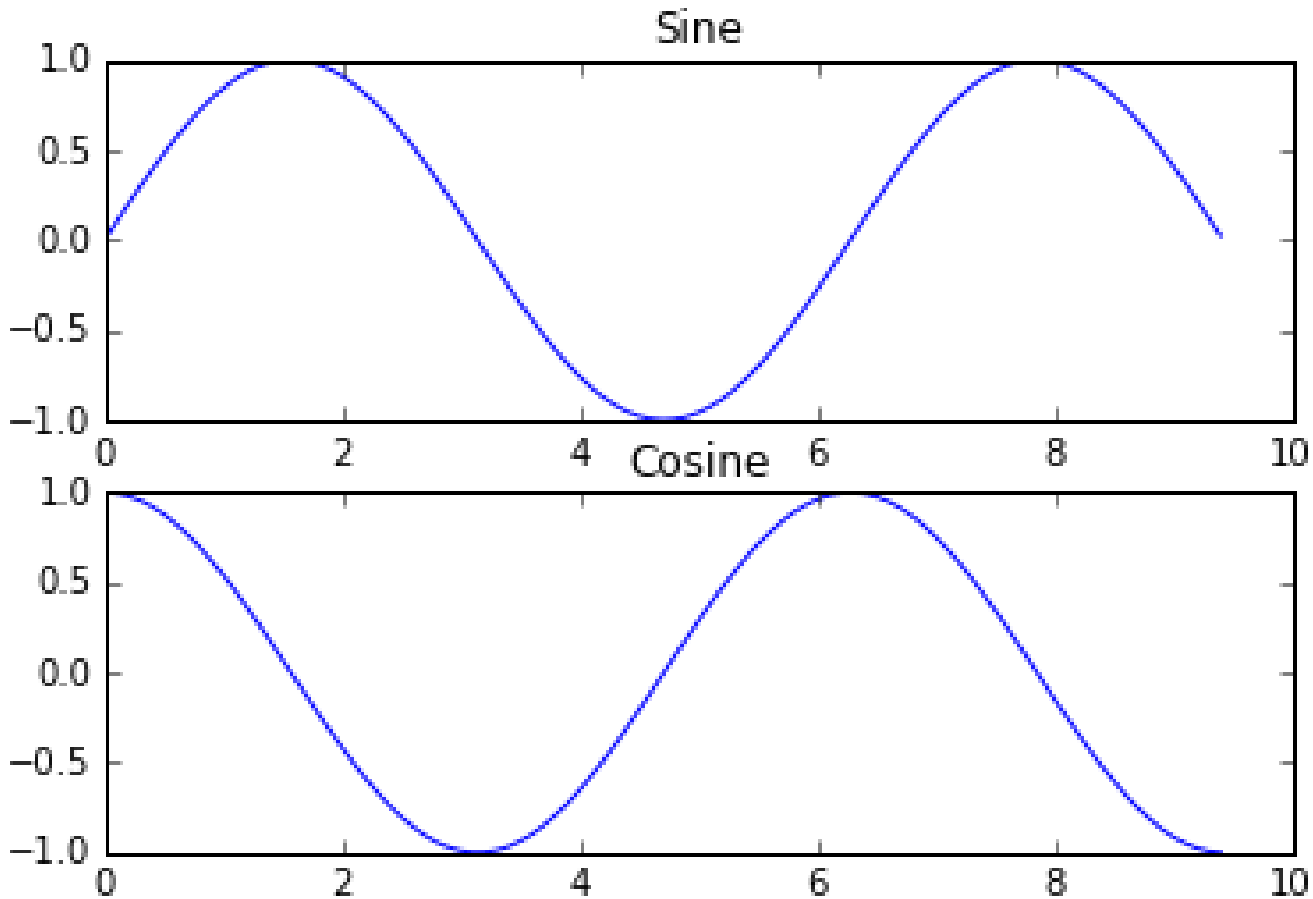
You can plot different things in the same figure using the subplot function.

```
# Compute the x and y coordinates
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)
# Set up a subplot grid: height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')
```

Subplots

```
# Set the second subplot as active
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

Plotting



Subplots

It is also possible to create a plot using categorical variables.

```
names = ['group_a', 'group_b', 'group_c']
values = [1, 10, 100]

plt.figure(1, figsize=(9, 3))

plt.subplot(131)
plt.bar(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.plot(names, values)
plt.suptitle('Categorical Plotting')
plt.show()
```

Plotting