# Programming for Business Computing

# Classes and Plotting

Ling-Chieh Kung

Department of Information Management
National Taiwan University

# Outline

- **Basics of classes**
- Advances of classes
- Plotting with **`matplotlib.pyplot`**

# Motivating example: dates

- In many applications we deal with dates.
  - Suppose that we do not know the **`datetime`** library.
- A date is consist of three attributes.
  - Year: an integer from 1 to 3000.
  - Month: an integer from 1 to 12.
  - Day: an integer from 1 to 31 (depending on month).
- If we want to store the birthdays of a group of students, what should we do?

# Birthday dictionary

```
bdDict = dict()
while True:
  name = input("name: ")
  if(name == ""):
    break

  birthday = input("birthday (yyyy/mm/dd): ")
  if(birthday == ""):
    break

  bdDict[name] = birthday

print(bdDict)
```

- How to prevent a date like 2016/14/20 or 2015/09/31?
  - Be aware of leap years!

# Is it a leap year?

- A year is a leap year if:
  - It is a multiple of 4 and not a multiple of 100.
  - It is a multiple of 400.

```
def isLeap(year):
  if(year % 400 == 0):
    return True
  elif((year % 4 == 0) and (year % 100 != 0)):
    return True
  else:
    return False
```

# Is it a good date?

```
def isGoodDate(birthday): # birthday is a yyyy/mm/dd string
  year, month, day = birthday.split("/")
  year = int(year)
  month = int(month)
  day = int(day)

  if((1 <= year <= 3000) and (1 <= month <= 12)):
    daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if(isLeap(year) == True):
      daysInMonth[1] = 29
    if(1 <= day <= daysInMonth[month - 1]):
      return True
  return False
```

# Fail-safe birthday dictionary

```
bdDict = dict()
while True:
  name = input("name: ")
  if(name == ""):
    break

  birthday = input("birthday (yyyy/mm/dd): ")

  if(isGoodDate(birthday) == True):
    bdDict[name] = birthday
  else:
    print("bad date!")

  if(birthday == ""):
    break

print(bdDict)
```

# What if…

- This is good, but what if we want to know:
  - The number of people born in a given year?
  - The names of people born in a given month?
- It would be better (in many cases) if we store **three integers** instead of a string.
  - Especially when the above operations must be done frequently.
- Option 1: Three dictionaries whose keys are names and values are years, months, and days, respectively.
  - It is hard to use and easy to be **inconsistent**.
- Option 2: One dictionary whose key is name and value is a three-dimensional list (or tuple or dictionary).
  - It is non-intuitive and/or **inefficient**.
- Is there a more intuitive way?

# Self-defined data types: class

- It is all about data types.
  - We have basic data types like character, integer, float, Boolean, etc.
  - We have composite date types like string, list, tuple, dictionary, etc.
  - May we define a new data type to store dates?
- In Python, we define our own data type by defining a **class**.
  - In a class, we define attributes called **instance variables**.
  - An attribute is also called a **member** of a class.
  - A class can then be used to declare **objects** (variables whose type is a class).

# Defining a class and declaring an object

- To define a class, we use the keyword **`class`**:

```
class Date:
  year = 1
  month = 1
  day = 1
```

  - Inside the class definition block, we declare instance variables one by one.
- Then we may use the class to declare objects:
  - We use the **dot operator** to access a member:

```
d = Date()
print(d.month)
print(d.day)
print(d) # what is this?
```

# Modifying an object

- We may modify an object by **modifying any of its member**.

```
d = Date()
d.month = 12
d.day = 31
print(d.month, d.day)
```

- In fact, we may "declare a member" outside the class definition block:

```
d = Date()
d.month = 12
d.weekday = "Mon"
print(d.month, d.weekday)
```

- Do not do this!
  - Unless you really know what you are doing.

# Why classes and objects?

- The most obvious reason of using classes and objects is to **group multiple variables into one variable**.
  - Each variable has its **variable name**.
- Recall our birthday dictionary example and our hope to store three integers.
  - Option 1: Three dictionaries whose keys are names and values are years, months, and days, respectively.
  - Option 2: One dictionary whose key is name and value is a three-dimensional list (or tuple or dictionary).
  - Option 3: One dictionary whose key is name (a string) and value is birthday (a `Date`).
- Let's revise our program with the class **`Date`**.

# Revising the birthday dictionary (1/4)

```
class Date: # the basic setting
  year = 1
  month = 1
  day = 1

def isLeap(year): # not changed
  if(year % 400 == 0):
    return True
  elif((year % 4 == 0) and (year % 100 != 0)):
    return True
  else:
    return False
```

# Revising the birthday dictionary (2/4)

```
def isGoodDate(bDay): # bDay is a Date object
  if((1 <= bDay.year <= 3000) and (1 <= bDay.month <= 12)):
    daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if(isLeap(bDay.year) == True):
      daysInMonth[1] = 29
    if(1 <= bDay.day <= daysInMonth[bDay.month - 1]):
      return True
  return False

def strToDate(birthday): # birthday is a yyyy/mm/dd string
  d = Date()
  year, month, day = birthday.split("/")
  d.year = int(year)
  d.month = int(month)
  d.day = int(day)
  return d
```

# Revising the birthday dictionary (3/4)

```
bdDict = dict()
while True:
  name = input("name: ")
  if(name == ""):
    break

  birthday = input("birthday (yyyy/mm/dd): ") # birthday is a string
  birthday = strToDate(birthday) # now birthday is a Date

  if(isGoodDate(birthday) == True):
    bdDict[name] = birthday # now the value of a dictionary entry is a Date
  else:
    print("bad date!")

  if(birthday == ""):
    break

print(bdDict) # what will be printed out?
```

# Revising the birthday dictionary (4/4)

```
def printBdayDict(bdDict):
  for p in bdDict.keys():
    b = bdDict[p] # b is a Date
    print(p + " was born on " + str(b.year) + "/" + str(b.month) + "/" + str(b.day))

bdDict = dict()
while True:
  // omitted; see previous page

printBdayDict(bdDict)
```

# Summary

- In short:
  - An object is a collection of variables (sometimes objects).
  - Moreover, these variables have names.
  - More benefits are to be introduced.

# Outline

- Basics of classes
- **Advances of classes**
- Plotting with **`matplotlib.pyplot`**

# Motivating example

- "ubike.csv" contains the YouBike information for the 30 stations in Da-An district from 2015/9/7 to 2015/10/8.
  - The numbers of available bikes and empty slots of all stations are collected once per hour.
  - The latitudes and longitudes of all stations are collected.
  - And more.
- We may write programs to do a lot of analyses.
- Let's see how to use classes to **modularize** our program.
  - This is a demonstration of an important programming design philosophy: **object-oriented programming** (**OOP**).

# The class `Station`

- In our program, there are many stations.
  - Each station has its name, number of total spaces, latitude, and longitude.
  - The current status of a station is represented by the number of available bikes (assumed to be the number of total spaces minus empty spaces).
  - Each station may be considered as an object.
- Let's define a class `Station` with five attributes:

```
class Station:
    name = 0
    space = 0
    latitude = 0
    longitude = 0
    available = 0
```

# Member functions

- A station has its **attributes** represented by its **member variables**.
- A station also has some operations that may be applied on it.
  - We may "**replenish**" it by adding bikes to (or removing bikes from) it.
  - We may **calculate** its distance to another station.
  - We may **send** some bikes from it to another station.
- Beside member variables (attributes), we may also define **member functions** (**operations**) for a class.
- Before we define them, let's see how to use them (if they have been defined).

# Invoking member functions

- To invoke a member function, we also use the **dot operation**.

```
gg = Station()
gg.name = "Gong-Guan Exit 2"
gg.space = 30
gg.latitude = 25.01476
gg.longitude = 121.534538
gg.available = 0

gg.printStatus() # print the status
print(gg.status()) # return the status
gg.replenish(20) # add 20 bikes
print(gg) # what's this?
```

- When we invoke an object's member function, we call the object the **invoking object** and the function the **invoked member function**.

# Member function: printStatus()

- Let's start by defining a function printing a station's information.

```
class Station:
  name = 0
  space = 0
  latitude = 0
  longitude = 0
  available = 0

  def printStatus(self):
    print(self.name + ": " + str(self.available) + "/" + str(self.space))
```

- We define a function **printStatus()** inside the class **Station**.
- A member function's **first parameter** is always the **invoking object** itself.
  - Through this parameter, we access the invoking object's member variables and member functions.
- It does not need to be named as "self."

# Member function: `printStatus()`

```
class Station:
  name = 0
  space = 0
  latitude = 0
  longitude = 0
  available = 0

  def printStatus(self):
    print(self.name + ": " + str(self.available) + "/" + str(self.space))

gg = Station()
gg.name = "Gong-Guan Exit 2"
gg.space = 30
gg.latitude = 25.01476
gg.longitude = 121.534538

gg.printStatus() # Gong-Guan Exit 2: 0/30
```

# Invoking a member function

- Pay attention to the way of invoking a member function!

```
gg.printStatus() # good
printStatus(gg) # syntax error
gg.printStatus(gg) # syntax error
```

- If we execute **`printStatus(gg)`**:
  - "NameError: name 'printStatus' is not defined"
  - The function is a **member function** (which belongs to a class), not a **global function** (which belongs to everyone).
- If we execute **`gg.printStatus(gg)`**:
  - "TypeError: printStatus() takes exactly 1 argument (2 given)"
  - The invoking object is considered as the first argument (even if there is nothing inside the pair of parentheses).

# Member function: `replenish()`

- Let's replenish (positively or negatively) a station.
  - An input argument specifies the number of bikes intended to be "added" to the station (may be negative).
  - The number of available bikes and spaces should be checked.

```
class Station:
  // (member variables omitted)

  def replenish(self, num):
    if self.available + num <= self.space and self.available + num >= 0:
      self.available += num
    elif self.available + num > self.space:
      self.available = self.space
    else:
      self.available = 0
```

# Member function: `replenish()`

- Our programs will get bigger and more complicated.
  - To remind us about our program, we write **comments**.
- We write comments especially for **functions**.
  - To indicate the meanings of parameters.
  - To indicate the **pre-condition** and **post-condition**: What happens before and after the function is invoked.

```
// pre-condition: num is the number of biked intended to be added to the station
   // num > 0 means adding; num < 0 means removing
// post condition: the station's available number of bikes becomes
   // min(available + num, space) if num > 0 or
   // max(available + num, 0) if num < 0
def replenish(self, num):
  // (definition omitted)
```

# Member function: `replenish()`

```
gg = Station()
gg.name = "Gong-Guan Exit 2"
gg.space = 30
gg.latitude = 25.01476
gg.longitude = 121.534538

gg.printStatus() # Gong-Guan Exit 2: 0/30
gg.replenish(20)
gg.printStatus() # Gong-Guan Exit 2: 20/30
gg.replenish(13)
gg.printStatus() # Gong-Guan Exit 2: 30/30
gg.replenish(-50)
gg.printStatus() # Gong-Guan Exit 2: 0/30
```

# Member function: distance()

- We may calculate a station's distance to another station.

```
import math

def haversine(lat1, lon1, lat2, lon2):
  lon1, lat1, lon2, lat2 = map(math.radians, [lon1, lat1, lon2, lat2])
  dlon = lon2 - lon1
  dlat = lat2 - lat1
  a = math.sin(dlat / 2) ** 2
  a += math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2) ** 2
  return 6367 * (2 * math.asin(math.sqrt(a)))

class Station:
  // (member variables omitted)

  def distance(self, s):
    return haversine(self.latitude, self.longitude, s.latitude, s.longitude)
```

# Member function: `sendBike()`

- We may send some bikes to another station.

```
class Station:
  // (member variables omitted)

  // pre-condition: self is the invoking station, s is another station,
     // num is the number of bikes intended to be sent from self to s
     // num > 0 means from self to s; num < 0 means from s to self
  // post-condition: self's number of available bikes will be deducted by
     // num according to the rule specified in replenish(); that of s will
     // be added by num according to the same rule
  def sendBike(self, s, num):
    self.replenish(-num)
    s.replenish(num)
```

  - Note that we may invoke another member function in a member function!

# Member function: `sendBike()`

```
gg = Station()
gg.name = "Gong-Guan Exit 2"
gg.space = 30
gg.latitude = 25.01476
gg.longitude = 121.534538

ntust = Station()
ntust.name = "NTUST"
ntust.space = 46
ntust.latitude = 25.0131
ntust.longitude = 121.539723
```

```
print(gg.distance(ntust))
print(ntust.distance(gg))

gg.replenish(20) # 20
gg.printStatus()
gg.replenish(13) # 30
gg.printStatus()

gg.sendBike(ntust, 18)
gg.printStatus() # 12
ntust.printStatus() # 18
```

# Member function: status()

- Let's define a function that returns a string of current status.

```
class Station:
  name = 0
  space = 0
  latitude = 0
  longitude = 0
  available = 0

  def status(self):
    return self.name + ": " + str(self.available) + "/" + str(self.space)

print(ntust.status())
```

- Why bother?
  - **status()** can be **a building block** of other tasks.
  - It better **modularizes** the program.

# Member function: `init()`

- In many cases, we want to avoid tedious and lengthy **member variable initialization** statements.

```
gg = Station()
gg.name = "Gong-Guan Exit 2"
gg.space = 30
gg.latitude = 25.01476
gg.longitude = 121.534538

ntust = Station()
ntust.name = "NTUST"
ntust.space = 46
ntust.latitude = 25.0131
ntust.longitude = 121.539723
```

- Let's write a member function that initializes member variables based on input arguments.

# Member function: `init()`

- Implementation:

```
class Station:
  // (member variables omitted)

  def init(self, name, space, latitude, longitude):
    self.name = name
    self.space = space
    self.available = 0
    self.latitude = latitude
    self.longitude = longitude

gg = Station()
gg.init("Gong-Guan Exit 2", 30, 25.01476, 121.534538)
gg.replenish(20)
gg.printStatus() # Gong-Guan Exit 2: 20/30
```

- Note how to distinguish a member variable and a parameter!

# Constructor: `__init__()`

- Even though we have defined **`init()`**, we cannot force one (ourselves) to invoke it.
  - We may play with **uninitialized objects**.
- To resolve this issue, Python (and many other languages) allows us to define a **constructor** for a class.
  - It is a member function called **`__init__()`**.
  - It is automatically invoked when an object is created.
  - The correct number of arguments must be prepared when creating an object.

# Constructor: `__init__()`

- Implementation:

```
class Station:
  // (member variables omitted)

  def __init__(self, name, space, latitude, longitude):
    self.name = name
    self.space = space
    self.available = 0
    self.latitude = latitude
    self.longitude = longitude

gg = Station("Gong-Guan Exit 2", 30, 25.01476, 121.534538)
gg.replenish(20)
gg.printStatus() # Gong-Guan Exit 2: 20/30

ntust = Station() # syntax error
```

# Printing out an object directly

- Recall the function **`status()`**, which returns a string of the current status.
  - We may print out the returned string.
  - However, we cannot **print out the object** directly.

```
class Station:
  // (member variables omitted)

  def status(self):
    return self.name + ": " + str(self.available) + "/" + str(self.space)

gg = Station("Gong-Guan Exit 2", 30, 25.01476, 121.534538)
print(gg.status()) # Gong-Guan Exit 2: 0/30
print(gg) # what is this?
```

# A special function: `__str__()`

- Let's change the name of **`status()`** to **`__str__()`** to see the magic:

```
class Station:
  // (member variables omitted)

  def __str__(self):
    return self.name + ": " + str(self.available) + "/" + str(self.space)

gg = Station("Gong-Guan Exit 2", 30, 25.01476, 121.534538)
print(gg) # Gong-Guan Exit 2: 0/30
```

- When we print out an object:
  - If a function named **`__str__()`** is defined, its returned value is printed out.
  - Otherwise, the memory information is printed out.

# Outline

- Basics of classes
- Advances of classes
- **Plotting with `matplotlib.pyplot`**

# Making plots with `matplotlib`

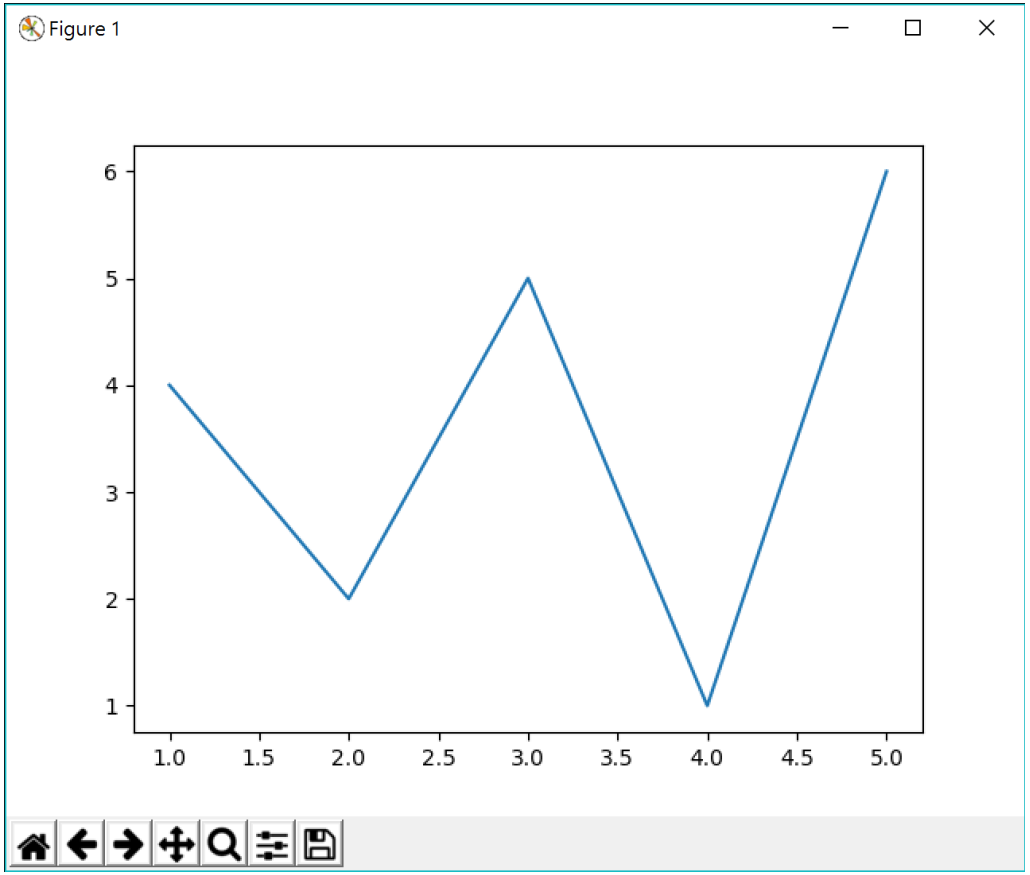
- In many cases, we want to make **plots**.
  - **Information visualization**.
- We will introduce how to use **`matplotlib`**, an open-source Python library, to make basic statistical plots.
  - Histograms, line charts, bar charts, pie charts, and scatter plots.
  - http://matplotlib.org/index.html
- First, we need to install **`matplotlib`**.
  - http://matplotlib.org/users/installing.html
  - Open your console/terminal/cmd, and execute:

```
pip install -U pip setuptools
pip install matplotlib
```

  - Do not forget to set the PATH variable or go to your Python script directory.

# Testing matplotlib

```
import matplotlib.pyplot as py

x = range(1, 6)
y = [4, 2, 5, 1, 6]
py.plot(x, y)
py.show()
```



# Data source

- To introduce the plotting tools in **`matplotlib`**, let's consider the data set contained in "midterm2.csv".

```
SubmissionID,StudentID,Problem,Status,Score,CodeLength,SubmissionTime
43629,7,4,Runtime Error,0,879,12:20:52
43628,31,3,Runtime Error,0,521,12:20:38
43627,106,2,Wrong Answer,0,10,12:20:27
43626,101,4,Wrong Answer,0,2330,12:20:27
43625,56,2,Wrong Answer,30,616,12:20:22
43624,13,2,Wrong Answer,0,1261,12:20:15
43623,84,2,Runtime Error,0,402,12:20:12
43622,78,2,Runtime Error,0,481,12:20:11
43621,31,3,Wrong Answer,0,521,12:20:11
43620,58,3,Wrong Answer,0,704,12:20:09
43619,46,2,Compile Error,0,1789,12:20:06
```

  - Student IDs are replaced by unique labels.

# Submission times

- We are interested in the students' **submission times**.
  - Is it true that students (altogether) make more submits when it is closer to the end of the exam?
- To answer this question, we may:
  - First, find the submission times, which is defined as the number of seconds since the exam starts (at 9:20:00) of a submission.
  - Second, draw a **histogram** for them: Number of submissions in [0, 1000), [1000, 2000), …, and [10000, 11000).

# Data processing

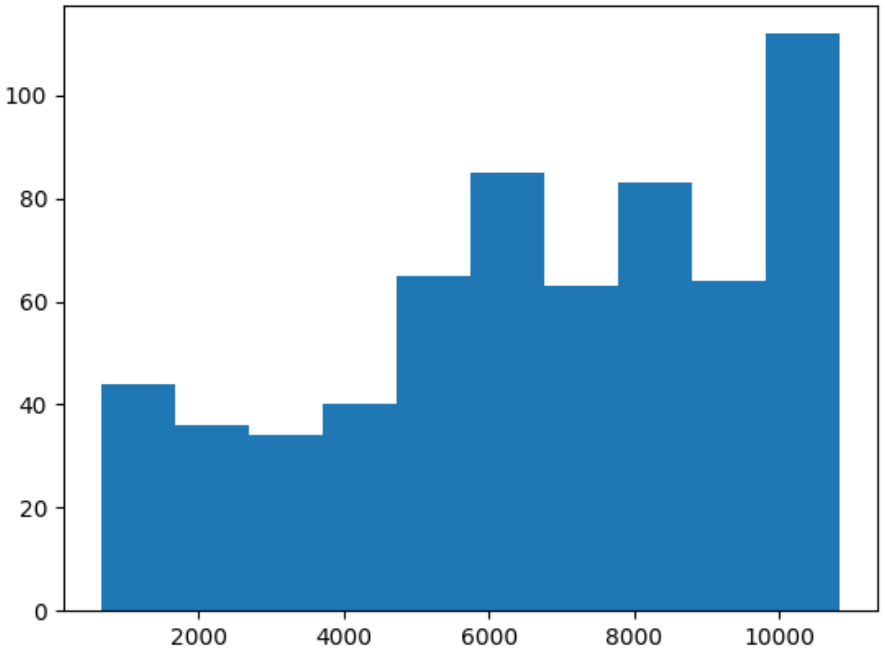
- First, we find the submission times:

```
import csv, datetime

def findSubTimes(fileName):
  fh = open(fileName, "r")
  csvFile = csv.DictReader(fh)
  next(csvFile)

  subTimes = [] # to store submission times
  for row in csvFile:
    dt = datetime.datetime.strptime(row["SubmissionTime"], "%H:%M:%S").time()
    sub = (dt.hour - 9) * 3600 + (dt.minute - 20) * 60 + dt.second
    subTimes.append(sub)

  fh.close()
  return subTimes

print(findSubTimes("midterm2.csv")) # just testing
```

# Making a histogram

- Second, we make a histogram:

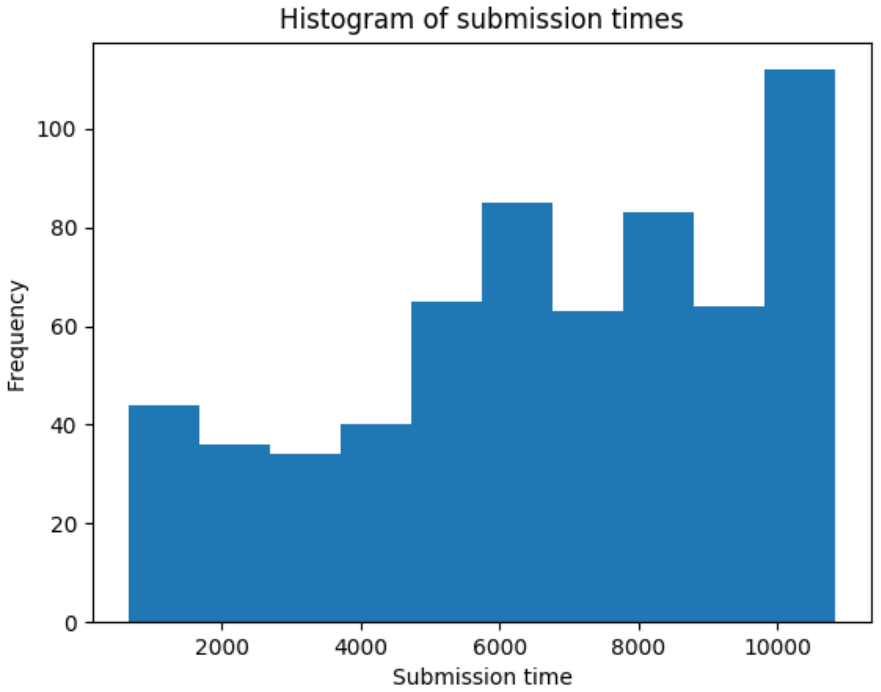
```
import matplotlib.pyplot as py

subTimes = findSubTimes("midterm2.csv")

py.hist(subTimes)
py.show()
```



# Decorating a histogram

- A plot should have *x*-label, *y*-label, and title (caption).

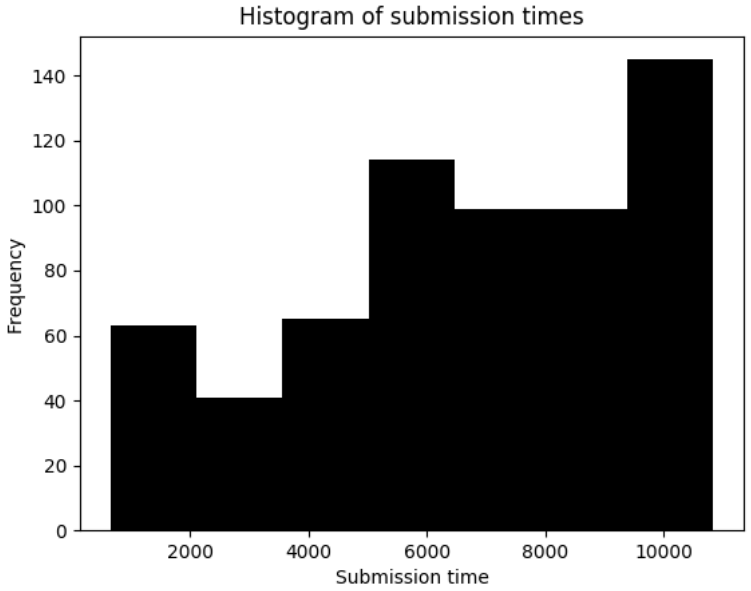
```
import matplotlib.pyplot as py

subTimes = findSubTimes("midterm2.csv")

py.hist(subTimes)
py.xlabel('Submission time')
py.ylabel('Frequency')
py.title('Histogram of submission times')
py.show()
```



# Setting up a histogram

- We may want to change bar colors and choose the number of classes.

```
import matplotlib.pyplot as py

subTimes = findSubTimes("midterm2.csv")

py.hist(subTimes, bins = 7, facecolor = "black")
py.xlabel('Submission time')
py.ylabel('Frequency')
py.title('Histogram of submission times')
py.show()
```





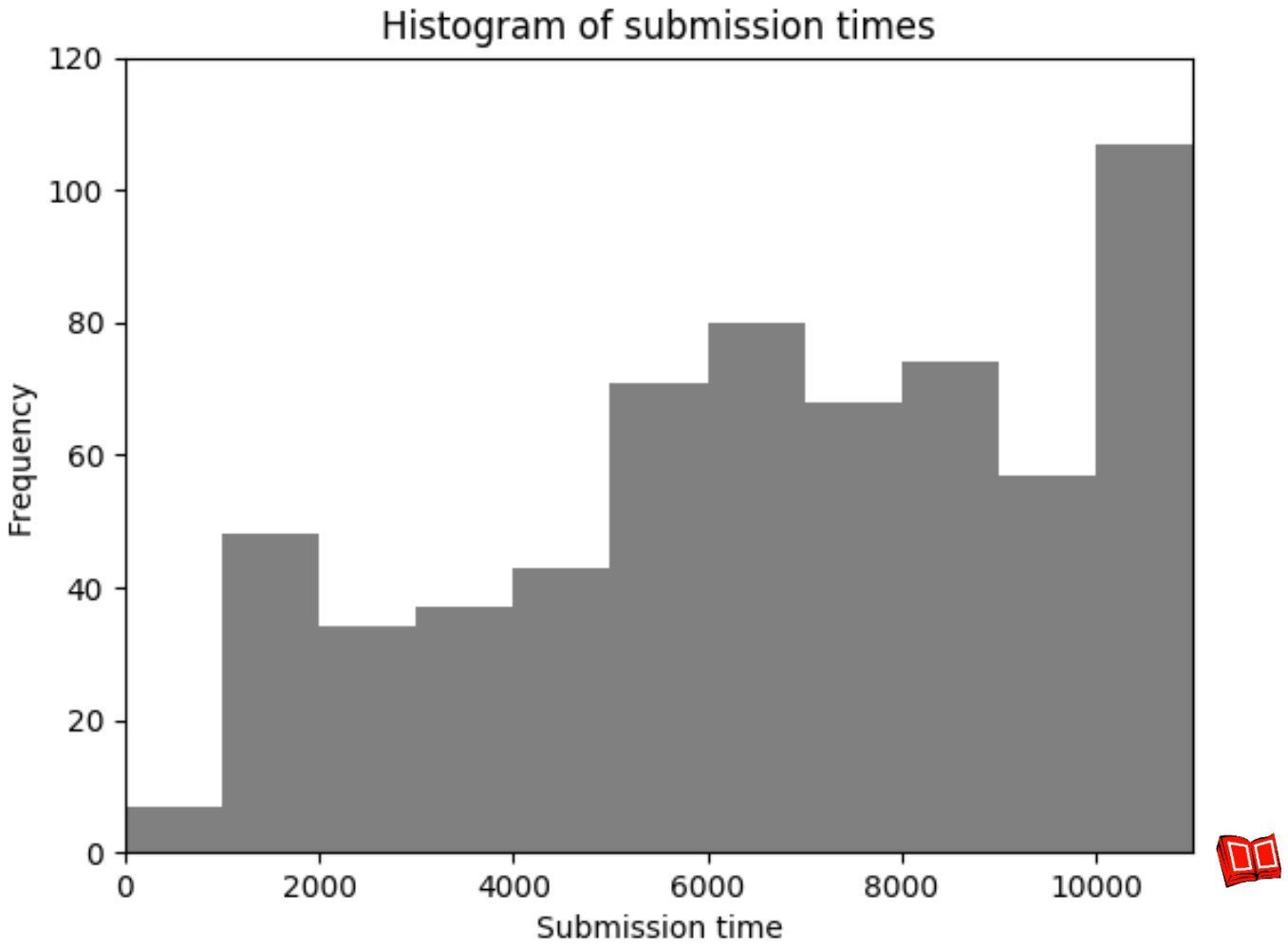

# Setting up a histogram

- We may **specify all classes** by ourselves.

```
import matplotlib.pyplot as py

subTimes = findSubTimes("midterm2.csv")

py.hist(subTimes, bins = range(0, 12000, 1000), facecolor = "gray")
py.ylim(0, 120)
py.xlim(0, 11000)
py.xlabel('Submission time')
py.ylabel('Frequency')
py.title('Histogram of submission times')
py.show()
```

# Setting up a histogram





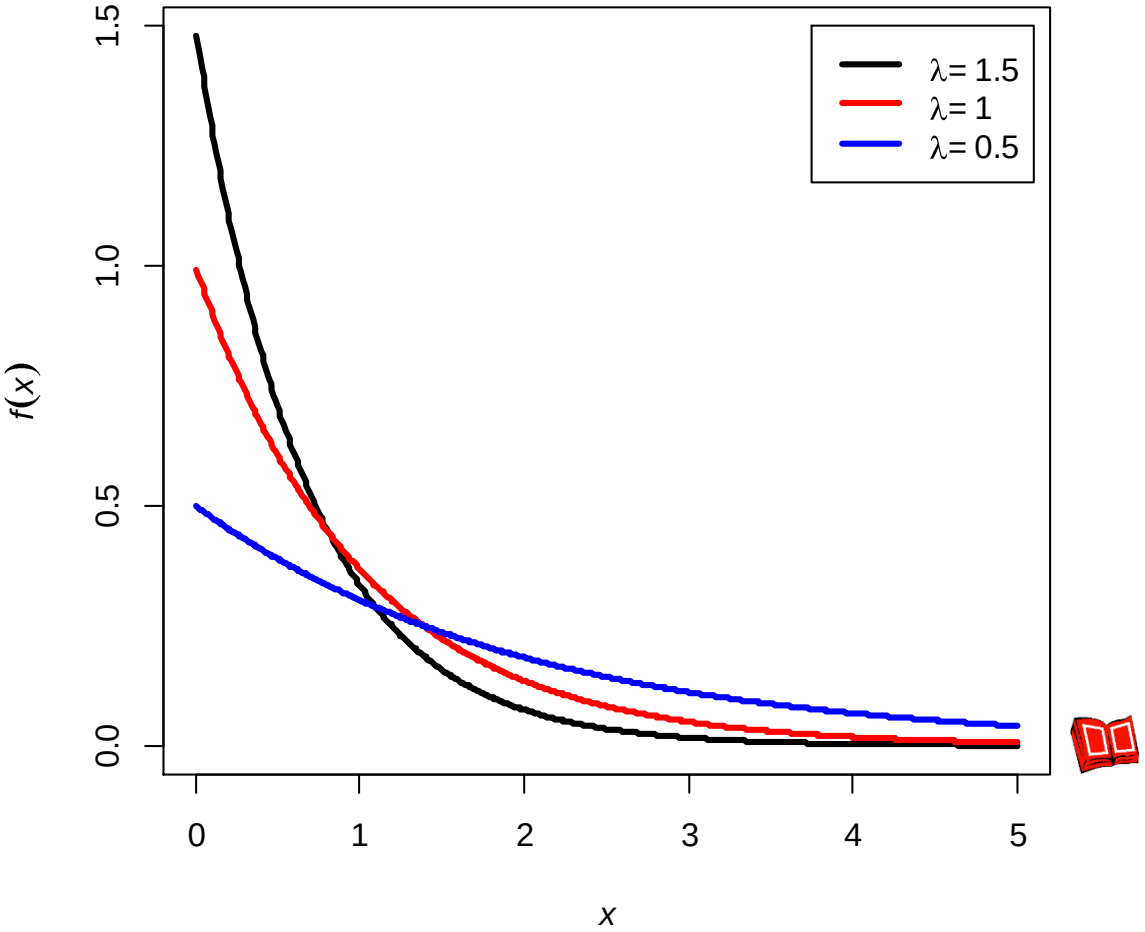
# Setting up a histogram

- We may want to obtain the frequencies (and/or class endpoints).

```
import matplotlib.pyplot as py

subTimes = findSubTimes("midterm2.csv")

n, bins, patches = py.hist(subTimes, bins = range(0, 12000, 1000))

print(n) # the frequencies
print(bins) # the class endpoints
```

# Inter-submission times

- We are interested in knowing the distribution of **inter-submission times**.
  - It is exponentially distributed?



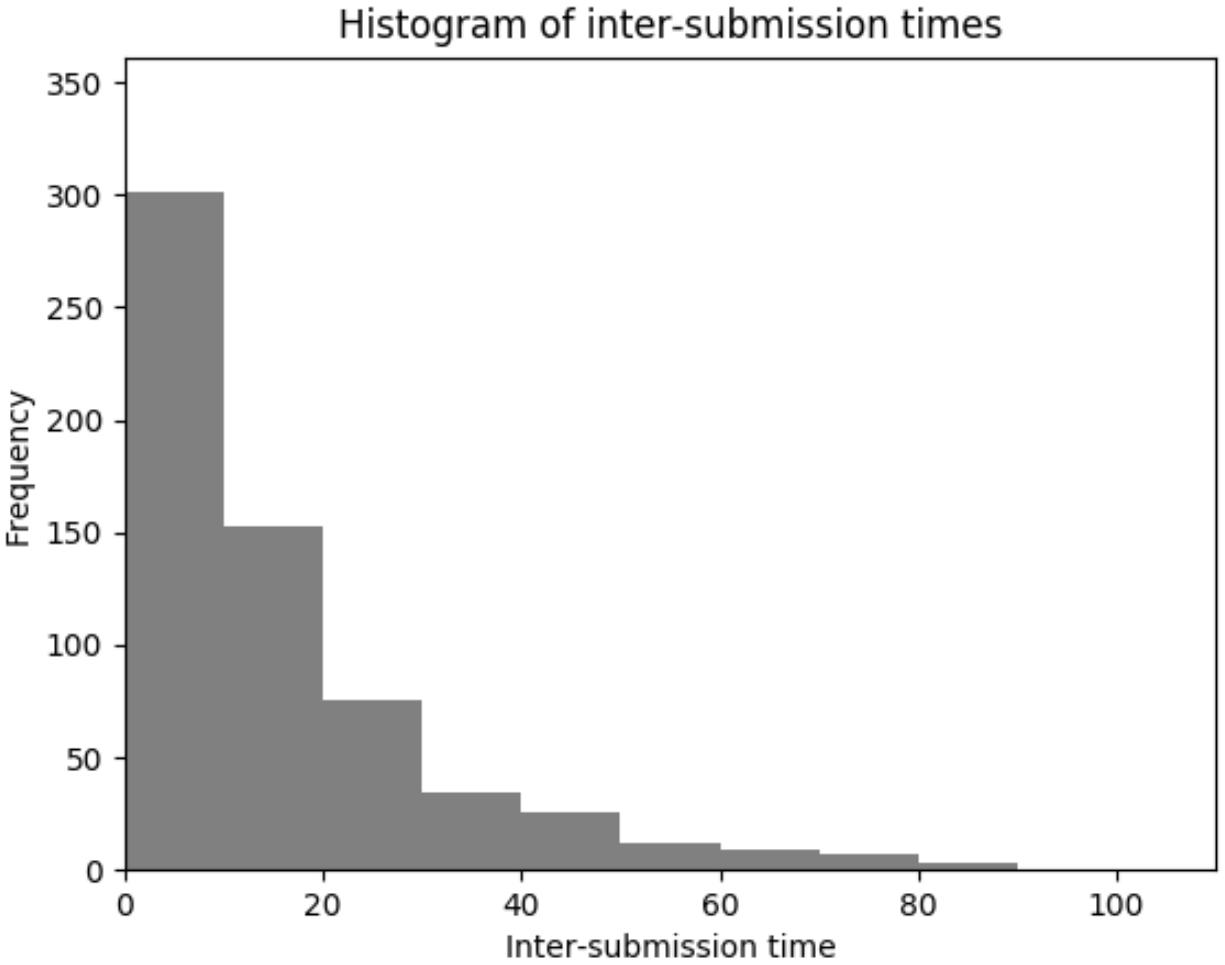
# Data processing

```
import csv, datetime

def findInterSubTimes(filePath):
  fh = open(filePath, "r")
  csvFile = csv.DictReader(fh)
  next(csvFile)

  interSubTimes = []
  preTime = 0

  for row in csvFile:
    dt = datetime.datetime.strptime(row["SubmissionTime"], "%H:%M:%S").time()
    sub = (dt.hour - 9) * 3600 + (dt.minute - 20) * 60 + dt.second
    if preTime > 0:
      interSubTimes.append(preTime - sub)
    preTime = sub

  fh.close()
  return interSubTimes
```

# Making a histogram

```
import matplotlib.pyplot as py

interSubTimes = findInterSubTimes("midterm2.csv")

py.hist(interSubTimes, bins = range(0, 110, 10), facecolor = "gray")
py.xlim(0, 110)
py.ylim(0, 100)
py.xlabel('Inter-submission time')
py.ylabel('Frequency')
py.title('Histogram of inter-submission times')
py.show()
```

# Making a histogram



# Each student's number of submissions

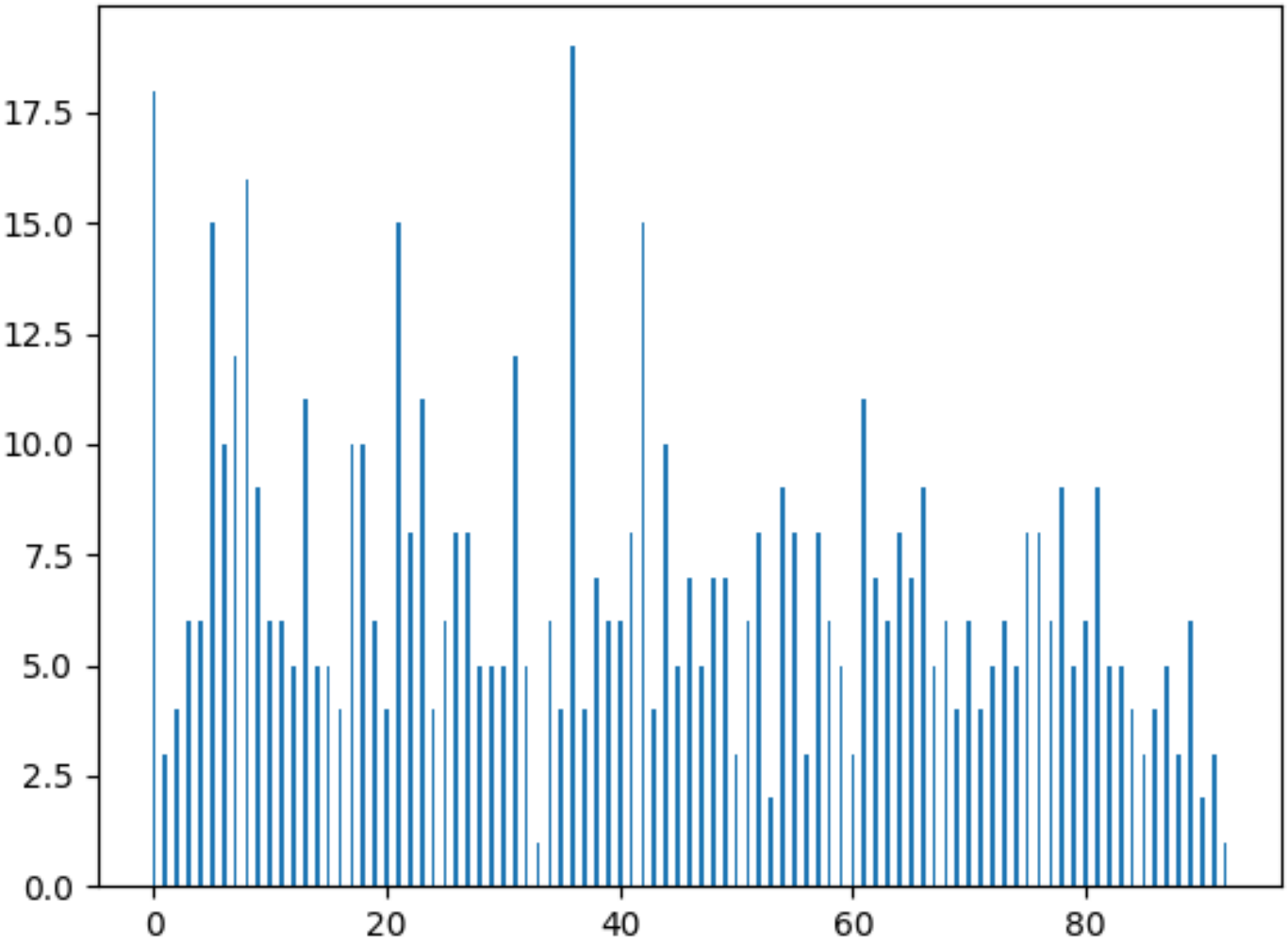
- Students may have different number of submissions.
- Beside using a histogram to visualize the distribution of the number of submissions, we may also use a **bar chart** to visualize all these numbers.

# Making a bar chart

```
import csv, datetime

def findSubFreq(filePath):
  fh = open(filePath, "r")
  csvFile = csv.DictReader(fh)
  next(csvFile)

  subFreqDict = dict()

  for row in csvFile:
    sid = int(row["StudentID"])
    if sid in subFreqDict:
      subFreqDict[sid] += 1
    else:
      subFreqDict[sid] = 1

  fh.close()

  return subFreqDict
```

```
import matplotlib.pyplot as py

subFreq = findSubFreq("midterm2.csv")

ind = range(0, len(subFreq))
width = 0.35
py.bar(ind, subFreq.values(), width)
py.show()
```

# Making a bar chart



# Making a pie chart

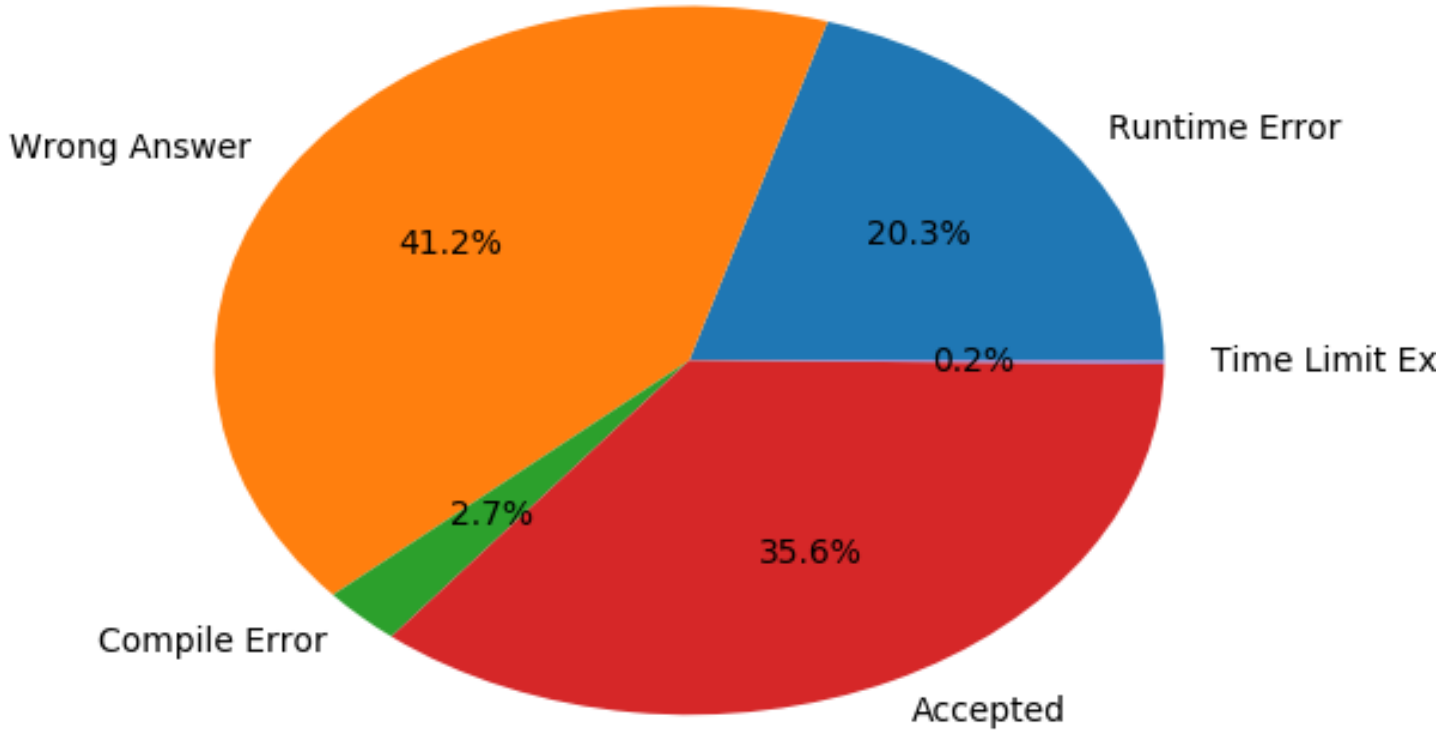
- When one submits, the result may be “Accepted”, “Wrong Answer”, “Runtime Error”, “Compile Error”, and “Time Limit Exceed”.
- We may use a **pie chart** to visualize their proportions.

# Making a pie chart

```
import csv, datetime

def findProp(filePath):
  fh = open(filePath, "r")
  csvFile = csv.DictReader(fh)
  next(csvFile)

  propDict = dict()

  for row in csvFile:
    result = row["Status"]
    if result in propDict:
      propDict[result] += 1
    else:
      propDict[result] = 1

  fh.close()

  return propDict
```

```
import matplotlib.pyplot as py

pf = findProp("midterm2.csv")

f = list(pf.values())
r = list(pf.keys())

py.pie(f, labels = r, autopct = "%1.1f%%")
py.show()
```

# Making a pie chart



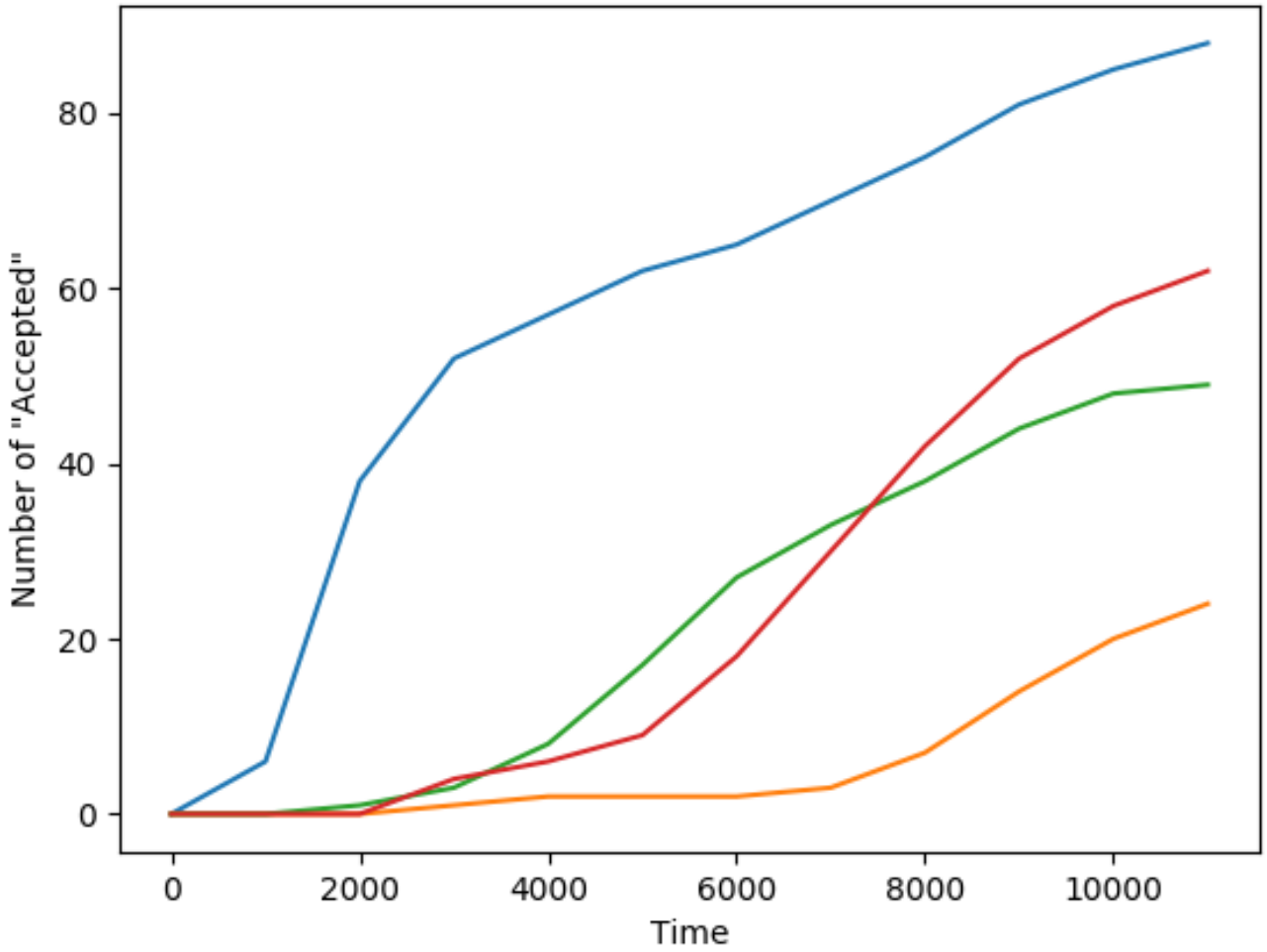
# Road to AC

- How was each problem be answered?
  - The speed of getting “Accepted”.
  - The difference between problems.
- We may draw a **line chart** and use four lines to represent the cumulative numbers of “Accepted” up to a certain time point, one for each problem.
- Note that this is not an easy task if MS Excel is the only tool!
  - How to process the data and calculate these numbers?

# Making a line chart (attempt 1)

```
import matplotlib.pyplot as py

p1 = [0, 6, 38, 52, 57, 62, 65, 70, 75, 81, 85, 88]
p2 = [0, 0, 0, 1, 2, 2, 2, 3, 7, 14, 20, 24]
p3 = [0, 0, 1, 3, 8, 17, 27, 33, 38, 44, 48, 49]
p4 = [0, 0, 0, 4, 6, 9, 18, 30, 42, 52, 58, 62]
times = range(0, 12000, 1000)

py.plot(times, p1)
py.plot(times, p2)
py.plot(times, p3)
py.plot(times, p4)
py.xlabel('Time')
py.ylabel('Number of "Accepted"')
py.show()
```

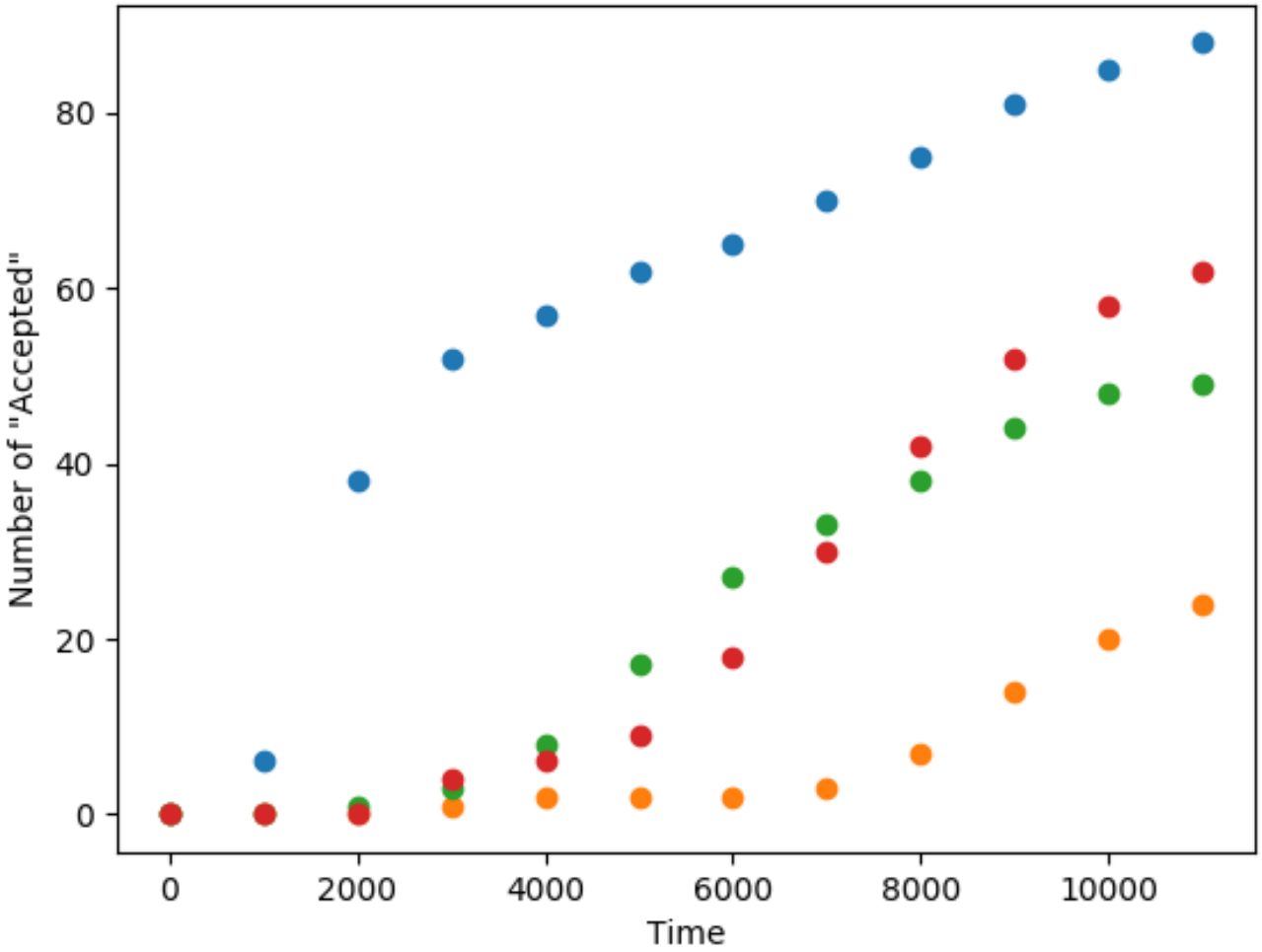
# Making a line chart (attempt 1)



# Making a line chart (attempt 2)

```
import matplotlib.pyplot as py

p1 = [0, 6, 38, 52, 57, 62, 65, 70, 75, 81, 85, 88]
p2 = [0, 0, 0, 1, 2, 2, 2, 3, 7, 14, 20, 24]
p3 = [0, 0, 1, 3, 8, 17, 27, 33, 38, 44, 48, 49]
p4 = [0, 0, 0, 4, 6, 9, 18, 30, 42, 52, 58, 62]
times = range(0, 12000, 1000)

py.plot(times, p1, 'o')
py.plot(times, p2, 'o')
py.plot(times, p3, 'o')
py.plot(times, p4, 'o')
py.xlabel('Time')
py.ylabel('Number of "Accepted"')
py.show()
```

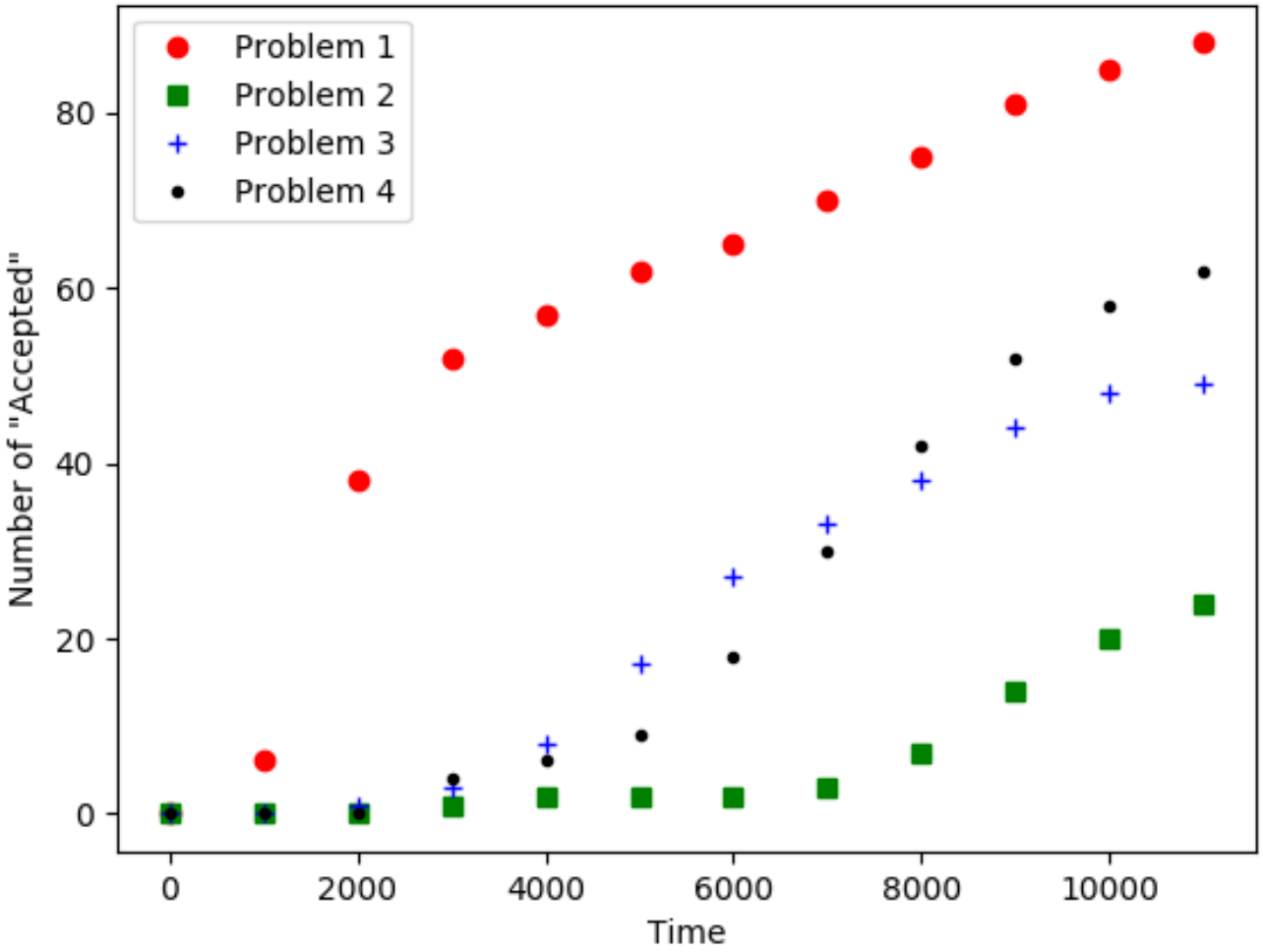
# Making a line chart (attempt 2)



# Making a line chart (attempt 3)

```
import matplotlib.pyplot as py

p1 = [0, 6, 38, 52, 57, 62, 65, 70, 75, 81, 85, 88]
p2 = [0, 0, 0, 1, 2, 2, 2, 3, 7, 14, 20, 24]
p3 = [0, 0, 1, 3, 8, 17, 27, 33, 38, 44, 48, 49]
p4 = [0, 0, 0, 4, 6, 9, 18, 30, 42, 52, 58, 62]

times = range(0, 12000, 1000)
py.plot(times, p1, 'ro', label = "Problem 1")
py.plot(times, p2, 'gs', label = "Problem 2")
py.plot(times, p3, 'b+', label = "Problem 3")
py.plot(times, p4, 'k.', label = "Problem 4")
py.legend(loc = 'upper left')
py.xlabel('Time')
py.ylabel('Number of "Accepted"')
py.show()
```

# Making a line chart (attempt 3)

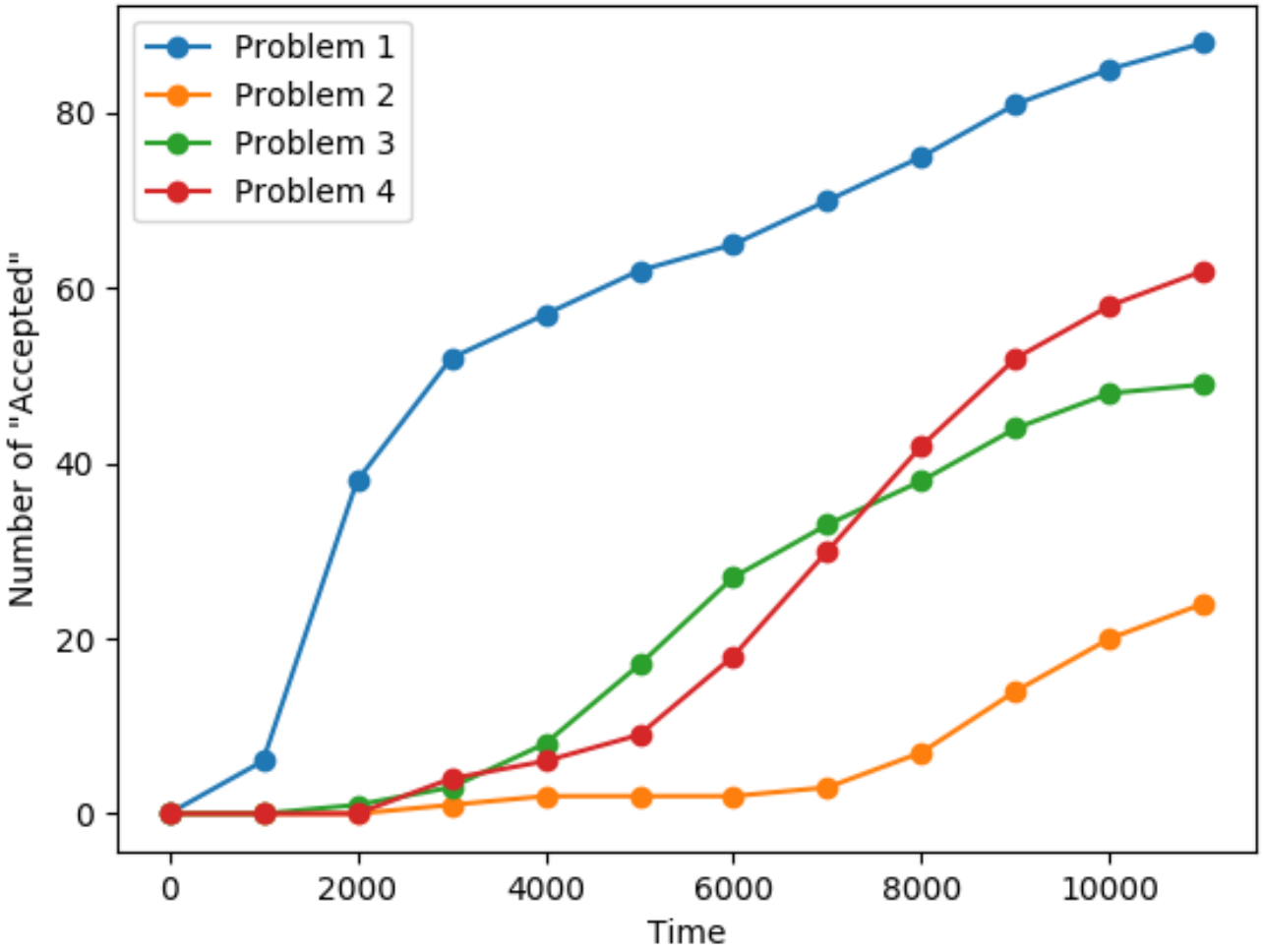





# Making a line chart (attempt 4)

```
import matplotlib.pyplot as py
p1 = [0, 6, 38, 52, 57, 62, 65, 70, 75, 81, 85, 88]
p2 = [0, 0, 0, 1, 2, 2, 2, 3, 7, 14, 20, 24]
p3 = [0, 0, 1, 3, 8, 17, 27, 33, 38, 44, 48, 49]
p4 = [0, 0, 0, 4, 6, 9, 18, 30, 42, 52, 58, 62]

times = range(0, 12000, 1000)
py.plot(times, p1, label = "Problem 1", marker = 'o')
py.plot(times, p2, label = "Problem 2", marker = 'o')
py.plot(times, p3, label = "Problem 3", marker = 'o')
py.plot(times, p4, label = "Problem 4", marker = 'o')
py.legend(loc = 'upper left')
py.xlabel('Time')
py.ylabel('Number of "Accepted"')
py.show()
```

# Making a line chart (attempt 4)

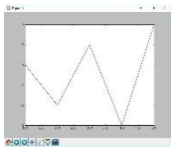
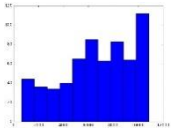
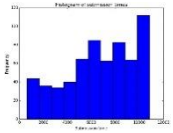
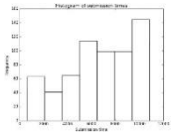
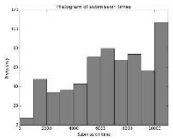




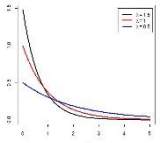
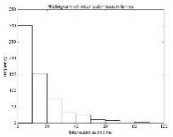
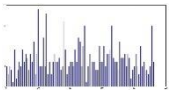
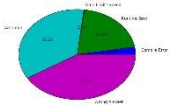
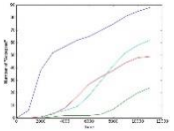
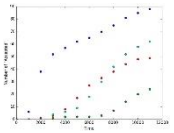

# Summary

- Use classes to organize and modularize your program.
  - Comments are important!
- Process data and visualize them with libraries.
  - Almost everything you want have been implemented and put somewhere on the Internet.
  - Search, copy, modify, try, understand, and create!

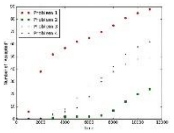
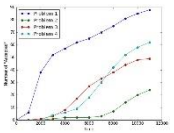
# 版權聲明

| 序 | 頁 | 作品 | 版權標章 | 作者 / 來源 |
|---|---|---|---|---|
| 1 | 1-73 |  |  | 台灣大學 孔令傑, CC BY-NC-ND 3.0 |
| 2 | 41 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 3 | 45 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 4 | 46 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 5 | 47 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 6 | 49 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |

# 版權聲明

| 序 | 頁 | 作品 | 版權標章 | 作者 / 來源 |
|---|---|---|---|---|
| 7 | 51 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 8 | 54 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 9 | 57 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 10 | 60 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 11 | 63 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 12 | 65 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |

# 版權聲明

| 序 | 頁 | 作品 | 版權標章 | 作者 / 來源 |
|---|---|---|---|---|
| 13 | 67 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| 14 | 69 |  |  | 2002 - 2012 John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team; 2012 - 2016 The Matplotlib development team. Last updated on May 10, 2017. Created using Sphinx 1.5.5.<br>依據著作權法第46、52、65條及Matplotlib版權聲明合理使用<br>Matplotlib only uses BSD compatible code, and its license is based on the PSF license.<br>https://matplotlib.org/<br>2017/9/27 visited |
| | | | | |
| | | | | |
| | | | | |
| | | | | |