9. Files and Exceptions

Objectives

- Understand the notions of files and persistent data.
- Read, write and update files.
- Read and write CSV files, a common format for machine-learning datasets.
- Serialize objects into the JSON data-interchange format—commonly used to transmit over the Internet—and deserialize JSON into objects.
- Use the `with` statement to ensure that resources are properly released, avoiding “resource leaks.”
- Use the `try` statement to delimit code in which exceptions may occur and handle those exceptions with associated `except` clauses.

Objectives (cont.)

- Use the `try` statement’s `else` clause to execute code when no exceptions occur in the `try` suite.
- Use the `try` statement’s `finally` clause to execute code regardless of whether an exception occurs in the `try`.
- `raise` exceptions to indicate runtime problems.
- Understand the traceback of functions and methods that led to an exception.
- Use pandas to load into a `DataFrame` and process the Titanic Disaster CSV dataset.

Outline

9.1 Introduction

- Variables, lists, tuples, dictionaries, sets, arrays, pandas `Series` and pandas `DataFrame`s offer only *temporary* data storage
  - lost when a local variable "goes out of scope" or when the program terminates
- Data maintained in **files** is persistent
- Computers store files on secondary storage devices
  - solid-state drives, hard disks and more

9.1 Introduction (cont.)

- We consider text files in several popular formats
  - plain text
  - JSON (JavaScript Object Notation)
  - CSV (comma-separated values)
- Use JSON to serialize and deserialize objects for saving to secondary storage and transmitting them over the Internet
- Intro to Data Science section shows loading and manipulating CSV data with
  - Python Standard Library's `csv` module
  - pandas Library

9.1 Introduction (cont.)

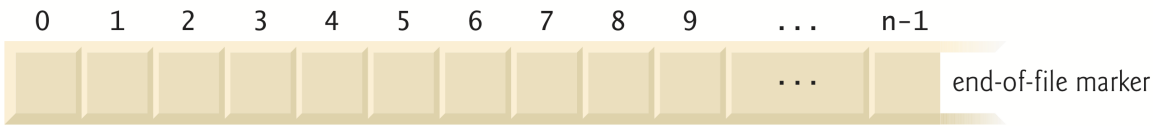
- Python security—we'll discuss the security vulnerabilities of serializing and deserializing data with the Python Standard Library's `pickle` module
  - We recommend JSON serialization in preference to `pickle`
- **exception handling**
  - An exception indicates an execution-time problem
  - E.g., `ZeroDivisionError`, `NameError`, `ValueError`, `StatisticsError`, `TypeError`, `IndexError`, `KeyError` and `RuntimeError`
  - Deal with exceptions as they occur by using `try` statements and `except` clauses
  - Help you write *robust*, *fault-tolerant* programs that can deal with problems and continue executing or *terminate gracefully*

9.1 Introduction (cont.)

- Programs typically request and release resources (such as files) during program execution
- Often, these are in limited supply or can be used only by one program at a time
- You'll see how to guarantee that after a program uses a resource, it's released for use by other programs, even if an exception has occurred
- Use the `with` statement for this purpose

9.2 Files

- A **text file** is a sequence of characters
- A **binary file** (for images, videos and more) is a sequence of bytes
- First character in a text file or byte in a binary file is located at position 0
  - In a file of ***n*** characters or bytes, the highest position number is ***n* – 1**
- For each file you **open**, Python creates a **file object** that you'll use to interact with the file



End of File

- Every operating system provides a mechanism to denote the end of a file
  - Some use an **end-of-file marker**
  - Others maintain a count of the total characters or bytes in the file
  - Programming languages hide these operating-system details from you

Standard File Objects

- When a Python program begins execution, it creates three **standard file objects**:
  - **`sys.stdin`** —the **standard input file object**
  - **`sys.stdout`** —the **standard output file object**, and
  - **`sys.stderr`** —the **standard error file object**.
- Though considered file objects, they do not read from or write to files by default
  - The `input` function implicitly uses `sys.stdin` to get user input from the keyboard
  - Function `print` implicitly outputs to `sys.stdout`, which appears in the command line
  - Python implicitly outputs program errors and tracebacks to `sys.stderr`, which also appears in the command line
- Import the `sys` module if you need to refer to these objects explicitly in your code—this is rare

---

9.3 Text-File Processing

- Here, we'll create a simple text file that might be used by an accounts-receivable system to track the money owed by a company's clients
- We'll then read that text file to confirm that it contains the data
- For each client, we'll store
  - client's account number
  - last name
  - account balance owed to the company
- These data fields represent a client **record**
- Programmers must structure files to meet their applications' requirements

---

9.3.1 Writing to a Text File: Introducing the `with` Statement

- Many applications *acquire* resources
  - files, network connections, database connections and more
- Should *release* resources as soon as they're no longer needed
- Ensures that other applications can use the resources
- `with` statement
  - Acquires a resource and assigns its corresponding object to a variable
  - Allows the application to use the resource via that variable
  - Calls the resource object's **`close` method** to release the resource

In [1]:

```
with open('accounts.txt', mode='w') as accounts:
    accounts.write('100 Jones 24.98\n')
    accounts.write('200 Doe 345.67\n')
    accounts.write('300 White 0.00\n')
    accounts.write('400 Stone -42.16\n')
    accounts.write('500 Rich 224.62\n')
```

- Can also **write to a file with `print`**, which **automatically outputs a `\n`**, as in

```
print('100 Jones 24.98', file=accounts)
```

In [2]:

```
# macOS/Linux Users: View file contents
!cat accounts.txt
```

```
100 Jones 24.98
200 Doe 345.67
300 White 0.00
400 Stone -42.16
500 Rich 224.62
```

In [ ]:

```
# Windows Users: View file contents
!more accounts.txt
```

Built-In Function `open`

- Opens the file `accounts.txt` and associates it with a file object
- `mode` argument specifies the **file-open mode**
  - whether to open a file for reading from the file, for writing to the file or both.
- Mode `'w'` opens the file for *writing*, creating the file if it does not exist
- If you do not specify a path to the file, Python creates it in the current folder
- **Be careful**—opening a file for writing *deletes* all the existing data in the file
- By convention, the **`.txt` file extension** indicates a plain text file

Writing to the File

- `with` statement assigns the object returned by `open` to the variable `accounts` in the **`as` clause**
- `with` statement's suite uses `accounts` to interact with the file
  - file object's **`write` method** writes one record at a time to the file
- At the end of the `with` statement's suite, the `with` statement *implicitly* calls the file object's **`close`** method to close the file

---

9.3.2 Reading Data from a Text File

- Let's read `accounts.txt` sequentially from beginning to end

In [1]:

```
with open('accounts.txt', mode='r') as accounts:
    print(f'{"Account":<10}{"Name":<10}{"Balance":>10}')
    for record in accounts:
        account, name, balance = record.split()
        print(f'{account:<10}{name:<10}{balance:>10}')
```

```
Account   Name         Balance
100       Jones          24.98
200       Doe           345.67
300       White           0.00
400       Stone         -42.16
500       Rich          224.62
```

9.3.2 Reading Data from a Text File (cont.)

- If the contents of a file should not be modified, open the file for reading only
  - Prevents program from accidentally modifying the file
- Iterating through a file object, reads one line at a time from the file and returns it as a string
- For each `record` (that is, line) in the file, string method `split` returns tokens in the line as a list
  - We unpack into the variables `account`, `name` and `balance`

File Method `readlines`

- File object's **`readlines`** method also can be used to read an *entire* text file
- Returns each line as a string in a list of strings
- For small files, this works well, but iterating over the lines in a file object, as shown above, can be more efficient
  - Enables your program to process each text line as it's read, rather than waiting to load the entire file

Seeking to a Specific File Position

- While reading through a file, the system maintains a **file-position pointer** representing the location of the next character to read
- To process a file sequentially from the beginning *several times* during a program's execution, you must reposition the file-position pointer to the beginning of the file
  - Can do this by closing and reopening the file, or
  - by calling the file object's **`seek`** method, as in

```
file_object.seek(0)
```

- The latter approach is faster

9.4 Updating Text Files

- Formatted data written to a text file cannot be modified without the risk of destroying other data
- If the name `'White'` needs to be changed to `'Williams'` in `accounts.txt`, the old name cannot simply be overwritten
- The original record for `White` is stored as

```
300 White 0.00
```

- If you overwrite the name `'White'` with the name `'Williams'`, the record becomes

```
300 Williams00
```

- The characters beyond the second "`i`" in `'Williams'` overwrite other characters in the line
- The problem is that records and their fields can vary in size

9.4 Updating Text Files (cont.)

- To make the preceding name change, we can:
  - copy the records before `300 White 0.00` into a temporary file,
  - write the updated and correctly formatted record for account 300 to this file,
  - copy the records after `300 White 0.00` to the temporary file,
  - delete the old file and
  - rename the temporary file to use the original file's name.
- Requires processing *every* record in the file, even if you need to update only one record
  - More efficient when an application needs to update many records in one pass of the file

Updating accounts `.txt`

- Update the `accounts.txt` file to change account 300's name from `'White'` to `'Williams'` as described above:

In [1]:
```
accounts = open('accounts.txt', 'r')
```

In [2]:
```
temp_file = open('temp_file.txt', 'w')
```

In [3]:

```
with accounts, temp_file:
    for record in accounts:
        account, name, balance = record.split()
        if account != '300':
            temp_file.write(record)
        else:
            new_record = ' '.join([account, 'Williams', balance])
            temp_file.write(new_record + '\n')
```

Updating accounts `.txt` (cont.)

- This `with` statement manages two resource objects, specified in a comma-separated list after `with`
  - If the account is not `'300'`, we write `record` (which contains a newline) to `temp_file`
  - Otherwise, we assemble the new record containing `'Williams'` in place of `'White'` and write it to the file

In [4]:

```
# macOS/Linux Users: View file contents
!cat temp_file.txt
```

```
100 Jones 24.98
200 Doe 345.67
300 Williams 0.00
400 Stone -42.16
500 Rich 224.62
```

In [ ]:

```
# Windows Users: View file contents
!more temp_file.txt
```

`os` Module File-Processing Functions

- To complete the update, delete the old `accounts.txt` file, then rename `temp_file.txt` as `accounts.txt`

In [5]:

```
import os
```

In [6]:

```
os.remove('accounts.txt')
```

- Use the **`rename` function** to rename the temporary file as `'accounts.txt'`

In [7]:

```
os.rename('temp_file.txt', 'accounts.txt')
```

In [8]:

```
# macOS/Linux Users: View file contents
!cat accounts.txt
```

```
100 Jones 24.98
200 Doe 345.67
300 Williams 0.00
400 Stone -42.16
500 Rich 224.62
```

In [ ]:

```
# Windows Users: View file contents
!more accounts.txt
```

9.5 Serialization with JSON

- **JSON (JavaScript Object Notation)** is a text-based, human-and-computer-readable, data-interchange format used to represent objects as collections of name–value pairs.
- Preferred data format for transmitting objects across platforms.

JSON Data Format

- Similar to Python dictionaries
- Each JSON object contains a comma-separated list of **property names** and **values**, in curly braces.

```
{"account": 100, "name": "Jones", "balance": 24.98}
```

- JSON arrays, like Python lists, are comma-separated values in square brackets.

```
[100, 200, 300]
```

- Values in JSON objects and arrays can be:
  - **strings** in **double quotes**
  - **numbers**
  - JSON Boolean values **`true`** or **`false`**
  - **`null`** (like `None` in Python)
  - **arrays**
  - **other JSON objects**

Python Standard Library Module `json`

- **`json` module** enables you to convert objects to JSON (JavaScript Object Notation) text format
- Known as **serializing** the data
- Following dictionary, which contains one key–value pair consisting of the key `'accounts'` with its associated value being a list of dictionaries representing two accounts

In [10]:

```
accounts_dict = {'accounts': [
    {'account': 100, 'name': 'Jones', 'balance': 24.98},
    {'account': 200, 'name': 'Doe', 'balance': 345.67}]}
```

Serializing an Object to JSON

- Write JSON to a file
- `json` module's **`dump` function** serializes the dictionary `accounts_dict` into the file

In [11]:

```
import json
```

In [12]:

```
with open('accounts.json', 'w') as accounts:
    json.dump(accounts_dict, accounts)
```

- Resulting file contains the following text—reformatted slightly for readability:

```
{"accounts":
[{"account": 100, "name": "Jones", "balance": 24.98},
 {"account": 200, "name": "Doe", "balance": 345.67}]}
```

- JSON delimits strings with *double-quote characters*.

Deserializing the JSON Text

- `json` module's **`load` function** reads entire JSON contents of its file object argument and converts the JSON into a Python object
- Known as **deserializing** the data

In [13]:

```
with open('accounts.json', 'r') as accounts:
    accounts_json = json.load(accounts)
```

In [14]:

```
accounts_json
```

Out[14]:

```
{'accounts': [{'account': 100, 'name': 'Jones', 'balance': 24.98},
  {'account': 200, 'name': 'Doe', 'balance': 345.67}]}
```

In [15]:

```
accounts_json['accounts']
```

Out[15]:

```
[{'account': 100, 'name': 'Jones', 'balance': 24.98},
 {'account': 200, 'name': 'Doe', 'balance': 345.67}]
```

In [16]:

```
accounts_json['accounts'][0]
```

Out[16]:

```
{'account': 100, 'name': 'Jones', 'balance': 24.98}
```

In [17]:

```
accounts_json['accounts'][1]
```

Out[17]:

```
{'account': 200, 'name': 'Doe', 'balance': 345.67}
```

Displaying the JSON Text

- `json` module's **`dumps` function** (`dumps` is short for "dump string") returns a Python string representation of an object in JSON format
- Canbe used to display JSON in a nicely indented format
  - sometimes called "pretty printing"
- When call includes the `indent` keyword argument, the string contains newline characters and indentation for pretty printing
  - Also can use `indent` with the `dump` function when writing to a file

In [18]:

```
with open('accounts.json', 'r') as accounts:
    print(json.dumps(json.load(accounts), indent=4))
```

```
{
    "accounts": [
        {
            "account": 100,
            "name": "Jones",
            "balance": 24.98
        },
        {
            "account": 200,
            "name": "Doe",
            "balance": 345.67
        }
    ]
}
```

9.7 Additional Notes Regarding Files

- Table of file-open modes for text files
  - *Reading* modes raise a `FileNotFoundError` if the file does not exist
  - Each text-file mode has a corresponding binary-file mode specified with `b`, as in `'rb'` or `'wb+'`

Mode	Description
`'r'`	Open a text file for reading. This is the default if you do not specify the file-open mode when you call open.
`'w'`	Open a text file for writing. Existing file contents are *deleted*.
`'a'`	Open a text file for appending at the end, creating the file if it does not exist. New data is written at the end of the file.
`'r+'`	Open a text file reading and writing.
`'w+'`	Open a text file reading and writing. Existing file contents are *deleted*.
`'a+'`	Open a text file reading and appending at the end. New data is written at the end of the file. If the file does not exist, it is created.

Other File Object Methods

- **`read`**
  - For a text file, returns a string containing the number of characters specified by the method's integer argument
  - For a binary file, returns the specified number of bytes
  - If no argument is specified, the method returns the entire contents of the file
- **`readline`**
  - Returns one line of text as a string, including the newline character if there is one
  - Returns an empty string when it encounters the end of the file
- **`writelines`**
  - Receives a list of strings and writes its contents to a file
- Classes that Python uses to create file objects are defined in the Python Standard Library's **`io` module** (https://docs.python.org/3/library/io.html)

9.8 Handling Exceptions

- Various types of exceptions can occur when you work with files
- **`FileNotFoundError`**
  - Attempt to open a non-existent file for reading with the `'r'` or `'r+'` modes
- **`PermissionsError`**
  - Attempt an operation for which you do not have permission
  - Try to open a file that your account is not allowed to access
  - Create a file in a folder where your account does not have permission to write
- `ValueError`
  - Attempt to write to a file that has already been closed

9.8.1 Division by Zero and Invalid Input

Division By Zero

Recall that attempting to divide by `0` results in a `ZeroDivisionError`:

In [1]:

```
10 / 0
```

```
---------------------------------------------------------------------
---
ZeroDivisionError                         Traceback (most recent call l
ast)
<ipython-input-1-a243dfbf119d> in <module>
----> 1 10 / 0

ZeroDivisionError: division by zero
```

- Interpreter **raises an exception** of type `ZeroDivisionError`
- Exception in IPython
  - terminates the snippet,
  - displays the exception's traceback, then
  - shows the next `In []` prompt so you can input the next snippet
- Exception in a script terminates it and IPython displays the traceback

Invalid Input

- `int` raises a `ValueError` if you attempt to convert to an integer a string (like `'hello'`) that does not represent a number

In [2]:

```
value = int(input('Enter an integer: '))
```

```
---------------------------------------------------------------------
---
ValueError                                Traceback (most recent call l
ast)
<ipython-input-2-b521605464d6> in <module>
----> 1 value = int(input('Enter an integer: '))

ValueError: invalid literal for int() with base 10: 'hello'
```

9.8.2 `try` Statements

- Can *handle* exceptions so code can continue processing
- Following code uses exception handling to catch and handle (i.e., deal with) any `ZeroDivisionError`s and `ValueError`s that arise—in this case, allowing the user to re-enter the input

In [1]:

```
# dividebyzero.py
"""Simple exception handling example."""

while True:
    # attempt to convert and divide values
    try:
        number1 = int(input('Enter numerator: '))
        number2 = int(input('Enter denominator: '))
        result = number1 / number2
    except ValueError:  # tried to convert non-numeric value to int
        print('You must enter two integers\n')
    except ZeroDivisionError:  # denominator was 0
        print('Attempted to divide by zero\n')
    else:  # executes only if no exceptions occur
        print(f'{number1:.3f} / {number2:.3f} = {result:.3f}')
        break  # terminate the loop
```

```
You must enter two integers


Attempted to divide by zero


100.000 / 7.000 = 14.286
```

`try` Clause

- **`try` statements** enable exception handling
- **`try` clause** followed by a suite of statements that *might* raise exceptions

`except` Clause

- `try` clause's suite may be followed by one or more **`except` clauses**
- Known as *exception handlers*
- Each specifies the type of exception it handles

`else` Clause

- After the last `except` clause, an optional **`else` clause** specifies code that should execute only if the code in the `try` suite **did not raise exceptions**

Flow of Control for a `ZeroDivisionError`

- The point in the program at which an exception occurs is often referred to as the **raise point**
- When an exception occurs in a `try` suite, it terminates immediately
- If there are any `except` handlers following the `try` suite, program control transfers to the first one
- If there are no `except` handlers, a process called *stack unwinding* occurs (discussed later)
- When an `except` clause successfully handles the exception, program execution resumes with the `finally` clause (if there is one), then with the next statement after the `try` statement.

Flow of Control for a `ValueError`

Flow of Control for a Successful Division

- When no exceptions occur in the `try` suite, program execution resumes with the `else` clause (if there is one); otherwise, program execution resumes with the next statement after the try statement

---

9.9 finally Clause

The `finally` Clause of the `try` Statement

- `try` statement may have a `finally` clause after any `except` clauses or the `else` clause
- **`finally`** clause is guaranteed to execute
  - In other languages, this makes the `finally` suite ideal for resource-deallocation code
  - In Python, we prefer the `with` statement for this purpose

Example

In [1]:

```
try:
    print('try suite with no exceptions raised')
except:
    print('this will not execute')
else:
    print('else executes because no exceptions in the try suite')
finally:
    print('finally always executes')
```

```
try suite with no exceptions raised
else executes because no exceptions in the try suite
finally always executes
```

In [2]:

```
try:
    print('try suite that raises an exception')
    int('hello')
    print('this will not execute')
except ValueError:
    print('a ValueError occurred')
else:
    print('else will not execute because an exception occurred')
finally:
    print('finally always executes')
```

```
try suite that raises an exception
a ValueError occurred
finally always executes
```

Combining `with` Statements and `try…except` Statements

- Most resources that require explicit release, such as files, network connections and database connections, have potential exceptions associated with processing those resources
- *Robust* file-processing code normally appears in a `try` suite containing a `with` statement to guarantee that the resource gets released

In [3]:

```
open('gradez.txt')  # non-existent file
```

```
---------------------------------------------------------------------
----
FileNotFoundError                         Traceback (most recent call l
ast)
<ipython-input-3-5b552683292b> in <module>
----> 1 open('gradez.txt')  # non-existent file

FileNotFoundError: [Errno 2] No such file or directory: 'gradez.txt'
```

In [4]:

```
try:
    with open('gradez.txt', 'r') as accounts:
        print(f'{"ID":<3}{"Name":<7}{"Grade"}')
        for record in accounts:
            student_id, name, grade = record.split()
            print(f'{student_id:<3}{name:<7}{grade}')
except FileNotFoundError:
    print('The file name you specified does not exist')
```

```
The file name you specified does not exist
```

9.10 Explicitly Raising an Exception

- Sometimes you might need to write functions that raise exceptions to inform callers of errors that occur
- **`raise`** statement explicitly raises an exception

```
raise ExceptionClassName
```

- Creates an object of the specified exception class
- Exception class name may be followed by parentheses containing arguments to initialize the exception object—typically a custom error message string
- Code that raises an exception first should release any resources acquired before the exception occurred
- It's recommended that you use one of Python's many built-in exception types (https://docs.python.org/3/library/exceptions.html)

9.12.1 Python Standard Library Module `csv`

- **`csv` module** provides functions for working with CSV files

Writing to a CSV File

- `csv` module's documentation recommends opening CSV files with the additional keyword argument `newline=''` to ensure that newlines are processed properly

In [1]:

```
import csv
```

In [2]:

```
with open('accounts.csv', mode='w', newline='') as accounts:
    writer = csv.writer(accounts)
    writer.writerow([100, 'Jones', 24.98])
    writer.writerow([200, 'Doe', 345.67])
    writer.writerow([300, 'White', 0.00])
    writer.writerow([400, 'Stone', -42.16])
    writer.writerow([500, 'Rich', 224.62])
```

- **`.csv` file extension** indicates a CSV-format file
- **`writer` function** returns an object that writes CSV data to the specified file object
- `writer`'s **`writerow` method** receives an iterable to store in the file
- By default, `writerow` delimits values with commas, but you can specify custom delimiters

In [3]:

```
!cat accounts.csv
```

```
100,Jones,24.98
200,Doe,345.67
300,White,0.0
400,Stone,-42.16
500,Rich,224.62
```

- `writerow` calls above can be replaced with one **`writerows`** call that outputs a comma-separated list of iterables representing the records
- If you write data that contains commas in a given string, `writerow` encloses that string in double quotes to indicate a *single* value

Reading from a CSV File

- Read records from the file `accounts.csv` and display the contents of each record

In [4]:

```
with open('accounts.csv', 'r', newline='') as accounts:
    print(f'{"Account":<10}{"Name":<10}{"Balance":>10}')
    reader = csv.reader(accounts)
    for record in reader:
        account, name, balance = record
        print(f'{account:<10}{name:<10}{balance:>10}')
```

```
Account   Name         Balance
100       Jones          24.98
200       Doe           345.67
300       White            0.0
400       Stone         -42.16
500       Rich          224.62
```

- `csv` module's **`reader` function** returns an object that reads CSV-format data from the specified file object
- Can iterate through the `reader` object one record of comma-delimited values at a time

---

9.12.2 Reading CSV Files into Pandas `DataFrames`

- Here, we demonstrate pandas' ability to load files in CSV format, then perform some basic data-analysis tasks

Datasets

- Enormous variety of free datasets available online
- **Rdatasets repository** provides links to over 1100 free datasets in comma-separated values (CSV) format

> https://vincentarelbundock.github.io/Rdatasets/datasets.html (https://vincentarelbundock.github.io/Rdatasets/datasets.html)

- **`pydataset` module** specifically for accessing Rdatasets

> https://github.com/iamaziz/PyDataset (https://github.com/iamaziz/PyDataset)

- Another large source of datasets is

> https://github.com/awesomedata/awesome-public-datasets (https://github.com/awesomedata/awesome-public-datasets)

- A commonly used machine-learning dataset for beginners is the **Titanic disaster dataset**

Working with Locally Stored CSV Files

- File we'll process in this example

In [1]:

```
!cat accounts.csv
```

```
100,Jones,24.98
200,Doe,345.67
300,White,0.0
400,Stone,-42.16
500,Rich,224.62
```

- Load a CSV dataset into a `DataFrame` with the pandas function **`read_csv`**
- `names` argument specifies the `DataFrame` 's column names
  - Without this argument, `read_csv` assumes that the CSV file's first row is a comma-delimited list of column names

In [2]:

```
import pandas as pd
```

In [3]:

```
df = pd.read_csv('accounts.csv',
                 names=['account', 'name', 'balance'])
```

In [4]:

```
df
```

Out[4]:

	account	name	balance
0	100	Jones	24.98
1	200	Doe	345.67
2	300	White	0.00
3	400	Stone	-42.16
4	500	Rich	224.62

- To save a `DataFrame` to a file using CSV format, call `DataFrame` method **`to_csv`**
- `index=False` indicates that the row names (`0` – `4` at the left of the `DataFrame`'s output above are not written to the file
- Resulting file contains the column names as the first row

In [5]:

```
df.to_csv('accounts_from_dataframe.csv', index=False)
```

In [6]:

```
!cat accounts_from_dataframe.csv
```

```
account,name,balance
100,Jones,24.98
200,Doe,345.67
300,White,0.0
400,Stone,-42.16
500,Rich,224.62
```

9.12.3 Reading the Titanic Disaster Dataset

- Titanic disaster dataset is one of the most popular machine-learning datasets

Loading the Titanic Dataset via a URL

- Load the Titanic Disaster dataset directly from GitHub:

In [1]:

```
import pandas as pd
```

In [2]:

```
titanic = pd.read_csv('https://vincentarelbundock.github.io/Rdatasets/csv/carData/TitanicSurvival.csv')
```

Viewing Some of the Rows in the Titanic Dataset

- Dataset contains over 1300 rows, each representing one passenger
- For large datasets, displaying a `DataFrame` shows only the first 30 rows, followed by "…" and the last 30 rows
- View the first five and last five rows with `DataFrame` methods **`head`** and **`tail`**
  - Both return five rows by default

In [3]:

```
pd.set_option('precision', 2)  # format for floating-point values
```

In [4]:

```
titanic.head()
```

Out[4]:

	Unnamed: 0	survived	sex	age	passengerClass
0	Allen, Miss. Elisabeth Walton	yes	female	29.00	1st
1	Allison, Master. Hudson Trevor	yes	male	0.92	1st
2	Allison, Miss. Helen Loraine	no	female	2.00	1st
3	Allison, Mr. Hudson Joshua Crei	no	male	30.00	1st
4	Allison, Mrs. Hudson J C (Bessi	no	female	25.00	1st

In [5]:

```
titanic.tail()
```

Out[5]:

	Unnamed: 0	survived	sex	age	passengerClass
1304	Zabour, Miss. Hileni	no	female	14.5	3rd
1305	Zabour, Miss. Thamine	no	female	NaN	3rd
1306	Zakarian, Mr. Mapriededer	no	male	26.5	3rd
1307	Zakarian, Mr. Ortin	no	male	27.0	3rd
1308	Zimmerman, Mr. Leo	no	male	29.0	3rd

- pandas adjusts each column's width, based on the widest value in the column or based on the column name, whichever is wider
- The value in the age column of row `1305` is `NaN` (not a number), indicating a missing value in the dataset

Customizing the Column Names

- First column has a strange name (`'Unnamed: 0'`)
- Can clean that up by setting the column names

In [6]:

```
titanic.columns = ['name', 'survived', 'sex', 'age', 'class']
```

In [7]:

```
titanic.head()
```

Out[7]:

	name	survived	sex	age	class
0	Allen, Miss. Elisabeth Walton	yes	female	29.00	1st
1	Allison, Master. Hudson Trevor	yes	male	0.92	1st
2	Allison, Miss. Helen Loraine	no	female	2.00	1st
3	Allison, Mr. Hudson Joshua Crei	no	male	30.00	1st
4	Allison, Mrs. Hudson J C (Bessi	no	female	25.00	1st

9.12.4 Simple Data Analysis with the Titanic Disaster Dataset

- Can use pandas to perform some simple analysis.
- Calling `describe` on a `DataFrame` containing both numeric and non-numeric columns produces descriptive statistics *only for the numeric columns*
  - in this case, just the `age` column

In [8]:

```
titanic.describe()
```

Out[8]:

	age
count	1046.00
mean	29.88
std	14.41
min	0.17
25%	21.00
50%	28.00
75%	39.00
max	80.00

9.12.4 Simple Data Analysis with the Titanic Disaster Dataset (cont.)

- Discrepancy in the `count` (`1046`) vs. the dataset's number of rows (1309—the last row's index was `1308` when we called `tail`)
  - Only `1046` (the `count` above) of the records contained an age
  - Rest were *missing* and marked as `NaN`
- When performing calculations, Pandas *ignores missing data (`NaN`) by default*
- For the `1046` people with valid ages
  - average (`mean`) age was `29.88` years old
  - youngest passenger (`min`) was just over two months old (`0.17 * 12` is `2.04`)
  - oldest (`max`) was 80
  - Median age was `28` (indicated by the `50%` quartile)
  - `25%` quartile is the median age in the first half of the passengers (sorted by age)
  - `75%` quartile is the median of the second half of passengers

9.12.4 Simple Data Analysis with the Titanic Disaster Dataset (cont.)

- Let's say you want to determine some statistics about people who survived
- Can compare the `survived` column to `'yes'` to get a new `Series` containing `True/False` values, then use `describe` to summarize the results

In [9]:

```
(titanic.survived == 'yes').describe()
```

Out[9]:

```
count      1309
unique        2
top       False
freq        809
Name: survived, dtype: object
```

- For non-numeric data, `describe` displays different descriptive statistics:
  - `count` is the total number of items in the result
  - `unique` is the number of unique values (`2`) in the result— `True` (survived) and `False` (died)
  - `top` is the most frequently occurring value in the result
  - `freq` is the number of occurrences of the `top` value

9.12.5 Passenger Age Histogram

- Visualization helps you get to know your data
- Pandas has many built-in visualization capabilities that are implemented with Matplotlib
- To use them in Jupyter, first enable Matplotlib support
  - "inline" used only in Jupyter, not IPython interactive mode

In [10]:

```
%matplotlib inline
```

- A histogram visualizes the distribution of numerical data over a range of values
- A `DataFrame` 's **`hist`** method analyzes each numerical column's data and produces a separate histogram for each numerical column

In [11]:

```
histogram = titanic.hist()
```