The function `np.where(condition, a_true, a_false)` returns an array chosen from `a_true` and `a_false` based on the `condition`. Consider the following version of `square_positive()`:

```python
def square_positive(x: np.ndarray) -> np.ndarray:
    return np.where(x > 0, x ** 2, x)
```

This is vectorized so it can be applied to arrays and it is much more performant than a `for`-loop solution.

Box 3.2 Further Reading

- NumPy Developers (2024b), for a thorough introduction to NumPy
- NumPy Developers (2024a), for the API reference that describes NumPy classes, functions, and methods in detail

3.3 Input and Output

J4

A program's **input** and **output** refer to the the information provided to a program from without and the information the program produces. We have already seen examples of Python programs' output in the form of text printed to a console. Thus far, our programs have had no input because they have contained all the information they need.

In this section, we consider a few important types of Python input and output. In section 3.4, graphical output will be introduced.

3.3.1 User Input

A user can interact directly with a Python program via the built-in function `input(prompt)`. The `prompt` argument is printed to the console and the user can type in a response, finishing with the ⏎ key. For instance, consider the following program:

```python
import fractions  # Built-in module
response = input(  # Solicit user input
    "What are my chances? (Enter a fraction): "  # Prompt
)
if float(fractions.Fraction(response)) > 0.:
    print("So you're tellin' me there's a chance. YEAH!")
```

Running this program prints the following prompt:

```
What are my chances? (Enter a fraction): |
```

Suppose the user enters `1/1_000_000`. This is read by the input function and stored as a `str` in the variable `response`. A string can be cast to a fraction using the `fractions` module's `fractions.Fraction()` function. The fraction can be converted to a float with the `float()` function. In the case that the user enters `1/1_000_000`, the following would print to the console:

```
So you're tellin' me there's a chance. YEAH!
```

3.3.2 Text Files

A **text file** is a common type of input and output. Text files contain information in the form of lines of text. They are typically readable by a human, such that a text file can be viewed in a text file viewer or editor (e.g., Windows Notepad and TextEdit). While some text files have extension `.txt`, most do not. In fact, a Python script (with extension `.py`) is a text file.

3.3.2.1 Reading a Text File In Python, a text file can be opened and read with the following pattern:

```python
with open("filename") as f:  # Open a file for reading
    contents = f.read()  # Read and assign the entire contents
```

The built-in `open()` function takes the optional argument `mode`, which can have one of the following values:

- `"r"`: Read only (default)
- `"w"`: Write only (will overwrite an existing file)
- `"a"`: Append to an existing file
- `"r+"`: Read and write

For instance, suppose a file named `hamlet.txt` in the working directory has the following contents:

```
To die, to sleep;
To sleep: perchance to dream: ay, there's the rub;
For in that sleep of death what dreams may come
```

The following program reads the file and prints a line every 3 seconds:

```python
import time  # Built-in module
with open("hamlet.txt", mode="r") as f:
    contents = f.read().splitlines()  # List of lines
for line in contents:
    print(line)
    time.sleep(3)  # Delays for at least 3 seconds
```

At the end of the `with` block, the file is closed automatically, but the `contents` variable lives on.

Other methods for reading text files include `readline()` and `readlines()`.

3.3.2.2 Writing a Text File A text file can be written to a file opened in modes `"w"` (i.e., overwrite), `"a"` (i.e., append), and `"r+"` (i.e., read and write). Suppose one wanted to append the text "—Hamlet, act III, scene I" as a new line to `hamlet.txt`. The following would achieve this aim:

```python
attribution = "\n---Hamlet, act III, scene I"
with open("hamlet.txt", mode="a"):
    f.write(attribution)
```

Another useful method for writing to a file is the `writelines()` method, which writes a list of strings to the file.

3.3.3 JSON Files

JavaScript Object Notation (JSON) is a text file data format that is often used to store and share data in a form that is not specific to any programming language. JSON data types include:

- Numbers: signed decimal numbers (e.g., `4`, `8.0`, and `5e-3`)
- Strings: sequences of Unicode characters in double quotation marks (e.g., `"A string"`)
- Booleans: values of `true` and `false`
- Arrays: ordered collections of elements between brackets `[]`, comma-separated (e.g., `[3.1, 9, "foo"]`)
- Objects: unordered collections of key-value pairs between braces `{}`, comma-separated; for instance,

  ```json
  {
      "name": "Rick Sanchez",
      "occupation": "Scientist",
      "minimum age": 70,
      "grandchildren": ["Morty", "Summer"]
  }
  ```

A JSON file name conventionally ends with a `.json` extension.

The reading and writing of JSON files with Python can be done with the standard library `json` module, which can be loaded with the following statement:

```python
import json
```

3.3.3.1 Reading Python reads JSON types and converts them into similar Python types. The conversions are summarized in table 3.1.

Table 3.1: JSON to Python reading conversion.

From JSON	object	array	string	number (int)	number (real)	true	false	null
To Python	`dict`	`list`	`str`	`int`	`float`	`True`	`False`	`None`

Let's say a JSON file `rick.json` has the JSON object describing Rick Sanchez from above. We can load it in with the following code:

```python
with open("rick.json", "r") as f:
    data = json.load(f)
```

The base object is converted to a `dict` and assigned to the variable `data`. If we print `data` it will appear as follows:

```
{"grandchildren": ["Morty", "Summer"],
 "minimum age": 70,
 "name": "Rick Sanchez",
 "occupation": "Scientist"}
```

Here `"Rick Sanchez"` is a Python `str`, `70` is a Python `int`, and `["Morty", "Summer"]` is a Python `list` of `str`ings.

3.3.3.2 Writing When writing a JSON file, Python converts its own types to similar JSON types. The conversions are summarized in table 3.2.

Table 3.2: Python to JSON writing conversion.

From Python	`dict`	`list`, `tuple`	`str`	`int`, `float`	`True`	`False`	`None`
To JSON	object	array	string	number	true	false	null

Suppose we would like to write the following `data` `dict` to a JSON file:

```python
acceleration = 9.81
time = np.linspace(0, 1, 11)
velocity = acceleration * time
position = acceleration/2 * time ** 2
data = {
    "time": time.tolist(),
    "acceleration": acceleration,
    "velocity": velocity.tolist(),
    "position": position.tolist()
}
```

Note that we have used the `np.ndarray` method `tolist()` to convert the arrays to lists. This is necessary because Python cannot convert arrays directly to JSON. The following pattern will write the JSON file `kinematics.json`:

```python
with open("kinematics.json", "w") as f:
    json.dump(data, f)
```

The JSON file can now be shared or read into another program at a later time.

3.3.4 CSV Files

Another common format for sharing data is the venerable comma-separated value (CSV) text file. There are several flavors of CSV files, but most have a 2D tabular form with commas separating values (and columns) in a record (i.e., row) and newline characters separating records. There are similar delimiter-separated value text files that use delimiters other than commas to separate values; common delimeters include the tab character `\t` and the colon `:`.

Python has the standard library module `csv` for reading and writing such files. It is imported with the statement

```python
import csv
```

3.3.4.1 Reading Reading a CSV file `data.csv` to a `list` of `list`s can be achieved with the following code:

```python
with open("data.csv", 'r') as f:
    data = list(csv.reader(f, delimiter=","))
```

For files with other delimeters, the `delimiter` argument can be used to pass the appropriate delimiter. To convert the `list` `data` to a NumPy array, the usual `np.array()` function can be used.

3.3.4.2 Writing Although we usually prefer to write arrays to JSON files (section 3.3.3) or NumPy files (section 3.3.5),[3] occasionally we need to write a CSV file. The following code will write a `list` of `list`s to a CSV file:

```python
data = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]  #  Data to save
with open("data.csv", "w") as f:
    writer = csv.writer(f, delimiter=",")
    writer.writerows(data)
```

To write a NumPy array `A` to a CSV file, first convert it to a `list` with `A.tolist()`.

3.3.5 NumPy Input and Output

NumPy has its own file reading and writing capabilities. We consider only its array reading and writing capabilities because we find them the most useful.

3.3.5.1 The `.npy` File A NumPy array can be stored in a `.npy` file, which is a **binary file**, a file that is encoded and decoded differently than text files. A `.npy` file is usually more compact than a text file (e.g., JSON) storing the same array. Similarly, the time it takes to save and load a `.npy` file is usually much less than for

3. JSON is preferred for its capability of storing more complex data structures (e.g., dictionaries and deeply nested lists) and strict standardization. NumPy `.npy` and `.npz` files are preferred for their capability of storing multidimensional arrays, stricter standardization, and potential for compression.

a text file storing the same array. Furthermore, unlike for text files, no additional processing (e.g., converting to and from lists) is required for the reading and writing of .npy files. Therefore, it is often best to save and load arrays to .npy files instead of to text files. However, a text file, especially a JSON file, is the best choice for compatibility.

The following statements save a NumPy array A to a .npy file:

```
with open("A.npy", "wb") as f:
    np.save(f, A, allow_pickle=True)
```

The file was opened in "wb" or "write binary" mode; here the b is required to write binary files.

The allow_pickle argument, by default True, toggles the use of Python pickling (see section 3.3.6) for object arrays (i.e., NumPy arrays with nonnumeric objects). An object array cannot be saved without pickling.

The following statements load a NumPy array from a .npy file:

```
with open("A.npy", "rb") as f:
    A = np.load(f, allow_pickle=False)
```

The allow_pickle argument is by default False due to security and compatibility issues with loading pickled object arrays. Passing allow_pickle=False allows the loading of trusted object array .npy files.

3.3.5.2 The .npz File Multiple NumPy arrays can be saved to and loaded from a .npz file. The following statements save NumPy arrays A and B to a .npz file:

```
with open("data.npz", "wb") as f:
    np.savez(f, A=A, B=B)
```

The arguments following the file f of the np.savez() do not have to be named, but if they are the name is saved in the .npz file. Otherwise, default names are used.

The following statements load NumPy arrays A and B from a .npz file:

```
with np.load("data.npz", allow_pickle=False) as data:
    A = data["A"]
    B = data["B"]
```

Here we have temporarily loaded the data and accessed each array with dictionary syntax, assigning it to variables A and B that survive the with block.

In addition to np.savez(), there is the similar np.savez_compressed() function that attempts to compress the .npz file. For certain types of data, this can reduce the file size significantly at the cost of the compression time elapsed during saving and the decompression time elapsed during loading.

3.3.6 Pickle Files

The standard library module `pickle` can be used to save and load binary **pickle files**, conventionally with the extension `.pickle`. There are a few disadvantages to using pickle files instead of other methods described in this section for storing data:

- Pickle files are Python-specific
- Pickle files can depend on the version of Python used to create them
- Pickle files are a security risk, so do not load an untrusted pickle file
- Pickle files are not human-readable because they are binary
- Saving and loading pickle files is usually slower than NumPy for arrays

However, there are some distinct advantages as well:

- Many custom class objects can be pickled
- Pickle files are compact
- No external packages are required to save and load pickle files

We prefer to use them only when other methods will not work or are clumsy (e.g., when the object to be saved is an instance of a custom class).

The pickle module can be loaded with the following statement:

```
import pickle
```

3.3.6.1 Reading

A pickle file `data.pickle` can be loaded with the following code:

```
with open("data.pickle", "rb") as f:
    data = pickle.load(f)
```

If multiple objects were stored in the pickle file, additional calls to `pickle.load()` will load them in the order they were pickled. Often, we will assemble all objects to be pickled into a single object like a `tuple` or a `dict`, in which case each object can be given a name (key).

3.3.6.2 Writing

Objects `foo` and `bar` can be written to a pickle file `data.pickle` with the following code:

```
with open("data.pickle", "wb") as f:
    pickle.dump((foo, bar), f)  # Bundled into a tuple
```

Here we have bundled `foo` and `bar` into a single `tuple` so a single `pickle.load()` call returns all the data. Alternatively, we could have bundled them in a `dict` like `{"foo": foo, "bar": bar)`, or we could have simply called `pickle.dump()` multiple times to save multiple objects in the same file.

3.4 Introducing Graphics

SY

The graphical presentation of numerical data is perhaps the most important output of an engineering computing program. We must begin by understanding the purpose of graphics:

Graphics *reveal* data. (Tufte 2001; p. 13)

Data can, of course, be presented in other ways. Small sets of data are sometimes best presented in table format. However, most data is best presented visually.

Good graphics require careful design. The following list characterizes some aspects of a quality graphic (p. 13):

- It shows the data
- It draws the viewer to the data, not its presentation
- It presents the truth of the data with minimal distortion
- It presents lots of data in a small space
- It makes understandable large data sets
- It draws the viewer to compare pieces of the data set
- It presents the data in important levels of detail (broad and fine)
- It has a clear purpose: description, exploration, tabulation, or decoration
- It is closely integrated with accompanying descriptions of the data set

A good graphic is an explanation. For instance, it explains how one variable is related to another. In some cases, a causal explanation is suggested, as in figure 3.2, which suggests economic elites have much greater influence on policy adoption than do average citizens.

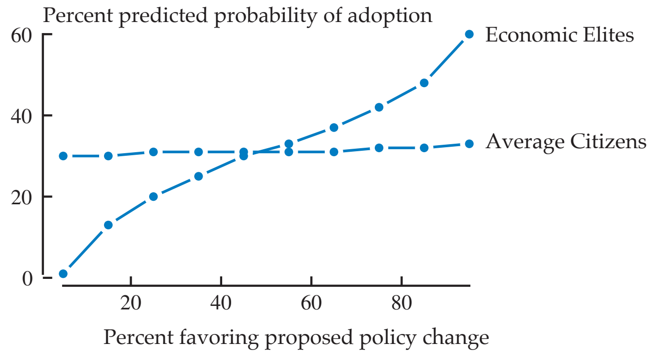

Figure 3.2. Percent predicted probability of public policy adoption for economic elites and average citizens. Study, results, and statistical model by Gilens and Page (2014).