Engineering Computing

Some Notes

Engineering Computing

Some notes

Rico A. R. Picone

© 2024 Rico A.R. Picone

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the author.

Contents

1	**Introduction**	**1**
	1.1 Introduction	1
	1.2 The Development System	2
	1.3 Basic Elements of a Program	6
	1.4 Lists	13
	1.5 Tuples and Ranges	16
	1.6 Dictionaries	17
	1.7 Functions	21
	1.8 Branching	22
	1.9 Looping	25
	1.10 Problems	27
2	**The Structure, Style, and Design of Programs**	**31**
	2.1 Python Interpreters and Interactive Sessions	31
	2.2 Scripts, Modules, and Imports	33
	2.3 The Python Standard Library and Packages	34
	2.4 Namespaces, Scopes, and Contexts	36
	2.5 Defining Classes	39
	2.6 Style Conventions	42
	2.7 The Design of Programs	46
	2.8 Problems	50
3	**Numerical Analysis I: Representations, Input and Output, and Graphics**	**55**
	3.1 Arrays	56
	3.2 Manipulating, Operating On, and Mapping Over Arrays	61
	3.3 Input and Output	67
	3.4 Introducing Graphics	74

3.5 Problems 83

4 Symbolic Analysis 91
4.1 Symbolic Expressions, Variables, and Functions 92
4.2 Manipulating Symbolic Expressions 95
4.3 Solving Equations Algebraically 107
4.4 From Symbolics to Numerics 117
4.5 Vectors and Matrices 122
4.6 Calculus 126
4.7 Solving Ordinary Differential Equations 130
4.8 Problems 136

5 Numerical Analysis II: Techniques 141
5.1 Problems 142

A Notebooks 143

B Documenting and Presenting Programs 145

C Version Control 147

D Lists of Figures and Tables 149
D.1 List of Figures 149
D.2 List of Tables 150

Bibliography 151

1 Introduction

In this chapter, we will get started with engineering computing by introducing the topic and setting up our tools to apply it throughout the rest of this book. Furthermore, the primary elements of the Python programming language are introduced.

1.1 Introduction

Engineering computing is the type of computing engineers use to design and analyze engineering systems. It is similar to **scientific computing**, and engineering and scientific computing share many tools, but the techniques and objectives of engineers differ from those of scientists. The computing tools of an engineer have become essential to the profession, and this has become increasingly true over the past few decades. Although the field can be considered inclusive of spreadsheet software and computer-aided design (CAD), we choose to focus on **computer programming**, writing text instructions called **programs** for a computer to perform calculations and store the results. We leave aside the important topic of **real-time computing** in which the computer becomes part of the engineering system; although it is increasingly important for engineers, it comes with a host of considerations that are a distraction from engineering computing.

There are many **computer languages**, but **programming languages** are those that are easiest for humans to use. Low-level programming languages give more control over the computer hardware and can be more compact; high-level languages have libraries and features that make programming easier. The most common programming languages used for engineering computing are the open-source **Python** language and the proprietary **MATLAB** language. Both are powerful languages with large userbases, but Python has been gaining in popularity in recent years. MATLAB has many built-in tools for engineering computation and "toolboxes" that extend its functionality beyond the base language. Python, on the other hand, does not have many built-in tools for engineering computation; however, it has

code libraries called **packages** that can be used for engineering computing. We will use a few key packages in this book, and there are many more available on the Internet, especially at https://engineering-computing.ricopic.one (Python Community 2024b).

There are several classes of engineering analysis performed with engineering computing. The following list captures the majority of problems, but it is far from exhaustive.

Numerical Analysis Many engineering problems can be approached by performing numerical calculations. These can be challenging or even intractable to perform manually when the problem requires many such calculations. **Numerical analysis** use systematic procedures called **algorithms** to perform the calculations with a computer. These techniques use the computer to perform, store, and organize these calculations. This class of problems, sometimes called **simulation**, comprise the majority of engineering computing problems.

Symbolic Analysis **Symbolic analysis**, sometimes called "analytic" as opposed to "numerical," is closely related to mathematics. Mathematical variables can be directly manipulated via algebraic methods (including those of calculus). Computer programs that treat these variables symbolically are called **computer algebra systems (CASs)**. Although these systems can be somewhat cumbersome, for complex problems they provide distinct advantages.

Graphical Analysis Visualization techniques are an important aspect of engineering analysis. **Graphics**—often graphs, plots, and charts—can be generated by programs much more quickly and accurately than they can be created manually. The result of an engineering computing program is often a graphic.

In this book, we will introduce all three classes of analysis.

1.2 The Development System

In general, a computer **development system** is one that is used to write, execute, debug, and deploy computer programs. Our development system is comprised of the following components:

- A personal computer (PC) (e.g., one running the Windows, macOS, or Linux operating system)
- The Anaconda distribution of the Python 3 software
- The **Spyder integrated development environment (IDE)**

An IDE is a software application in which a programmer can write, execute, and debug their programs.

On your PC, set up your development system with the following steps:

1. Download the Anaconda distribution of the Python software from the following URL:
   www.anaconda.com/download
   Open the installer and follow the instructions for installation.
2. Download and install the Spyder IDE from the following URL:
   www.spyder-ide.org
   Open the installer and follow the instructions for installation.

1.2.1 The Anaconda Distribution of Python

Anaconda provides a way of managing multiple **Python environments**; a Python environment is a specific version of Python with a set of packages. For a given project, it is best practice to maintain a separate environment; this allows us to specify a Python version and set of packages required to run the programs in the project. Anaconda provides a framework in which we can create an environment, called a **conda environment**.

We will use the default `base` environment. To create your own environments or add packages, see the instructions in the Anaconda documentation: https://engineering-computing.ricopic.one/0e.

0E

1.2.2 Hello World and the Spyder IDE

When it is first loaded, the Spyder IDE looks something like what is shown in figure 1.1.

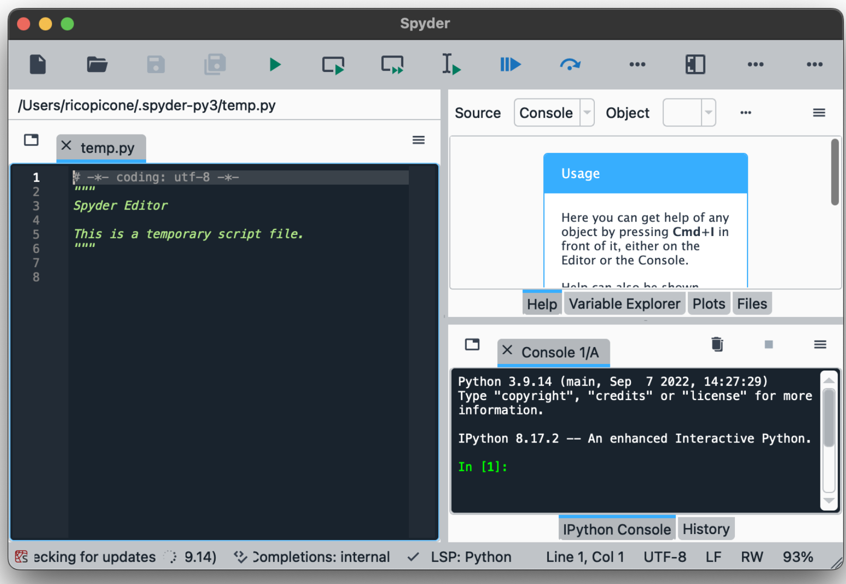


Figure 1.1. The Spyder IDE when it first loads

The left pane is the code editor. It has a default **Python file**, which conventionally has extension `.py`, already queued up. Create a new Python file by selecting the menu item File › New file… . Save this file (File › Save) as `hello_world.py` in a dedicated directory.[1]

The `hello_world.py` file already contains a triple-quoted string with basic information about the file. Below the ending quotes, add the following statement:

```
print("Hello World!")
```

Save the file and run it with the menu selection Run › Run or the key F5.

The **console** pane on the lower right shows the result of the execution of the file, which is the output

```
Hello World!
```

Now let's edit the program as follows:

1. In programming, file names should not include spaces, periods (other than for the extension), or most special characters. As a word separator, the hypen - is usually fine, but the underscore _ is topically safer. For Python files, the underscore is preferable.

```
greeting = "Hello World!"
print(greeting)
```

This should yield the same result in the console. In the upper-right pane, select the Variable Explorer tab. This shows variable names, types, and values in the current **kernel**. A kernel is a computing process that runs programs. In most environments, when a program runs, a kernel is created at the start and destroyed at the end of execution. However, Spyder maintains the same kernel between runs. This is convenient for debugging purposes. For instance, we can interact with the program(s) run in the current kernel by entering commands in the console; try entering

```
greeting
```

This will return the value of the variable `greeting`. The console is a convenient place to try out statements as we work on our program. For instance, we may want to append some text to the `greeting` string. In the console, try

```
greeting + "It's a beautiful day"
```

This returns, `Hello World!It's a beautiful day`, which is close but not quite what we wanted. We should add a space character to the beginning of our addendum. So, trying it out in the console allowed us to quickly debug our code.

The persistent kernel can also cause problems. Sometimes we may want to create a new kernel by selecting the menu item Consoles › Restart kernel, which clears all variables and unloads any packages. Similarly, to clear all variables in the kernel, we can execute the console **magic command**

```
%reset
```

We will be asked to confirm, which we can do by entering `y`.

1.2.3 Configuring the Spyder IDE for Anaconda

In section 1.2.2, we used the Python distribution that Spyder has built in. We here configure Spyder to use the Anaconda distribution installed in section 1.2.1. First, we must install the `spyder-kernels` package in the `base` Anaconda environment. On a Windows PC, open the Anaconda Prompt application; on MacOS or Linux, open the Terminal application. To ensure you have activated the `base` environment, enter the following prompt:

```
conda activate base
```

Now install the `spyder-kernels` package with the command

```
conda install spyder-kernels
```

Enter `y` if prompted. After successful installation, `conda list` should display the packages installed in the `base` environment, including `spyder-kernels`. Finally, enter the command

```
which python
```

Copy or record the returned path.

In Spyder, open preferences with Ctrl + , . Navigate to the tab Python Interpreter and check Use the following Python interpreter. Either paste the path copied above in the text field or click the Select file button, then navigate to the path in question, selecting the `python` program. Click OK to complete the configuration.

You may need to restart Spyder for the changes to take effect.

1.3 Basic Elements of a Program

Every programming language has a **syntax**: rules that describe the structure of valid combinations of characters and words in a program. When one first begins writing in a programming language, it is common to generate **syntax errors**, improper combinations of characters and words. Every programming language also has a **semantics**: a meaning associated with a syntactically valid program. A program's semantics describe what a program does.

In Python and in other programming languages, programs are composed of a sequence of smaller elements called **statements**. Statements do something, like perform a calculation or store a value in memory. For instance, `x = 3*5` is a statement that computes a product and stores the result under the variable name `x`. Many statements contain **expressions**, each of which produces a value. For instance, `3*5` in the previous statement is an expression that produces the value `15`.

An expression contains smaller elements called **operands** and **operators**. Common operands include **identifiers**—names like variables, functions, and modules that refer to objects—and **literals**—notations for constant values of a built-in type. For instance, in the previous expression `x` is a variable identifier and `3` and `5` are literals that evaluate to objects of the built-in `int`eger class. The `*` character in the previous expression is the multiplication operator. Python includes operators for arithmetic (e.g., `+`), assignment (e.g., `=`), comparison (e.g., `>`), logic (e.g., `or`), identification (e.g., `is`), membership (e.g., `in`), and other operations.

Example 1.1

Create a Python program that computes the following arithmetic expressions:

$$x = 4069 \cdot 0.002, \quad y = 100/1.5, \text{ and } \quad z = (-3)^2 + 15 - 3.01 \cdot 10.$$

Multiply these together (xyz) and print the product, along with x, y, and z to the console.

Consider the following program:

```
x = 4096*0.002              # float multiplication
y = 100/1.5                 # float division
z = (-3)**2 + 15 - 3.01*10  # exponent operator **
print(x,y,z)
print(x*y*z)
```

The console should print

```
8.192 66.66666666666667 -6.099999999999998
-3331.413333333333
```

Note that although we have multiplied and divided integer literals (`4096` and `100`) by floating-point literals (`0.002` and `1.5`), Python has automatically assumed we would like floating-point multiplication and division.

We used the exponent operator `**`, which may have been unfamiliar. If you tried the more common character ^ for the exponent, you received the error

```
TypeError: unsupported operand type(s) for ^: 'int' and 'float'
```

In Python, the ^ is the bitwise logical XOR operator.

1.3.1 Classes, Objects, and Methods

Everything that is expressed in a Python statement is an **object**, and every object is an **instance** of a **class**. For instance, `7` is a literal that evaluates to an object that is an instance of the `int`eger class. Similarly, `"foo"` is a literal that evaluates to an object that is an instance of the `str`ing class. A class can be thought of as a definition of the kind of objects that belong to it, how they are structured, and the kinds of things that can be done with it.

Python includes built-in classes such as the numeric `int`eger, `float`ing-point number, and `complex` number. It is common to refer to a class, especially a built-in class, as a **type**.

Classes are more than types of data, however. Classes can include one or more **method**, which is a kind of function that operates on inputs called **arguments** and returns outputs. Something special about methods is that they can operate on instances of the class. For example,

```
3.5.as_integer_ratio()
```

The literal `3.5` yields an instance of the `float` class, which has method `as_integer_ratio()`. Placing the `.` character before the method name is the syntax to apply the object's `as_integer_ratio()` method. This method returns a `tuple` object with the form `(<numerator>, <denominator>)`, where `<numerator>` and

`<denominator>` denote the numerator and denominator of the integer ratio corresponding to the floating-point number. The expression yields the output

```
(7, 2)
```

which signifies the fraction 7/2.

We can and often do create our own classes with their own methods. We will return to this topic in a later chapter.

Example 1.2

Create a Python program that starts with the three word strings `"veni"`, `"vedi"`, `"vici"` and concatenates and prints them with the following caveats:

- Between each word string, insert a comma and a space.
- Capitalize each word string using the `capitalize()` method.

Consider the following program:

```
w1 = "veni"
w2 = "vedi"
w3 = "vici"
print(w1.capitalize() + ", " +
    w2.capitalize() + ", " +
    w3.capitalize()
)
```

The console should print

```
Veni, Vedi, Vici
```

Note that we have used linebreaks to improve code readability. Python syntax allows expressions enclosed in parentheses to be broken after operators.

1.3.2 Basic Built-In Types

Python has several built-in types (classes) that provide a foundation from which many of our programs can be written. We have seen some examples of these types already, and in this section they will be described in greater detail.

1.3.2.1 Boolean The simple `bool` (i.e., Boolean) type can have one of two values, `True` or `False`. This type is used extensively for logical reasoning in programs, and will be especially important for branching (see section 1.8). Expressions containing the **Boolean operators** `not`, `and`, and `or` evaluate to Boolean values. For instance,

```
not True          # => False
not False         # => True
True and False    # => False
True or False     # => True
```

Similarly, expressions with the **comparison operators** `==` (equality), `!=` (inequality), `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), `is` (identity), `is not` (nonidentity), and `in` (membership). evaluate to Boolean values. A **truth table** for the Boolean operators and some comparison operators is given in table 1.1.

Table 1.1. Boolean and comparison operators on Boolean and integer inputs `x` and `y`

x	y	bool(x)	not x	x and y	x or y	x==y	x!=y	x<y	x<=y	x>=y	x>y
False	False	False	True	False	False	True	False	False	True	True	False
False	True	False	True	False	True	False	True	True	True	False	False
True	False	True	False	False	True	False	True	False	False	True	True
True	True	True	False	True	True	True	False	False	True	True	False
0	0	False	True	0	0	True	False	False	True	True	False
0	1	False	True	0	1	False	True	True	True	False	False
1	0	True	False	0	1	False	True	False	False	True	True
1	1	True	False	1	1	True	False	False	True	True	False

Note that non-Boolean inputs can be given to the Boolean operators. Non-Boolean objects can be given Boolean values with the `bool()` function, included in the table. For instance, `bool(0)` and `bool(0.0)` evaluate to `False`; conversely, `bool(1)` and `bool(1.0)` evaluate to `True`. In fact, for all numeric types (i.e., `int`, `float`, and `complex`), every value evaluates to `True` except those equivalent to `0`.

1.3.2.2 Integer The `int` (i.e., integer) type can be used to represent the mathematical integers, positive and negative (and 0). As we have already seen, several built-in operators can be applied to integer inputs, including `+` (summation), `-` (difference), `*` (product), and `/` (quotient). More operators will be introduced in later chapters.

The built-in `int()` function returns an integer representation of its input, which can be either a number, a string, or empty, in which case `int()` returns `0`. Although it does not round in the most elegant manner, `int()` can be used to convert a floating-point number to an integer, as in

```
int(3.2)     # => 3
int(3.9)     # => 3
int(-3.9)    # => -3
```

We see that `int()` rounds toward zero.

1.3.2.3 Floating-Point Number Like scientific notation, **floating-point numbers** represent potentially very large or very small numbers in a compact form. This form has three parts: a **sign** s, a **significand** x, and an **exponent** n. These combine as

$$s \times x \times 2^n.$$

Floating-point numbers can be represented with the Python `float` type and are often used to represent decimal numbers, such as 1.4 and −0.33. The built-in `float()` function returns a `float` from a number or string argument. For instance,

```
float(5)    # => 5.0
float("5")  # => 5.0
```

Floating-point numbers can be entered with scientific notation via the letter `e`, as in the following examples:

```
291e-6      # => 0.000291
1e3         # => 1000.0
```

1.3.2.4 Complex Number Complex numbers, which have a real part a and imaginary part b, represented mathematically as

$$a + jb,$$

where j is the imaginary number $\sqrt{-1}$, can be represented in Python with the `complex` type. For numbers `a` and `b`, we can construct a `complex` type with `complex(a, b)`. For instance,

```
complex(1, 2)   # => (1+2j)
```

A `complex` object has attributes `real` and `imag` that return the real and imaginary parts, respectively. For instance,

```
s = complex(3, -5)
s.real          # => 3
s.imag          # => -5
```

1.3.2.5 String Strings are series of characters and have built-in Python type `str`. String literals can be written with either single quotes (e.g., `'foo'`) or double quotes (e.g., `"bar"`). Within one variety, the other is treated as a regular quote, as in `"A 'friend' wants to know"` and `'I am "big boned"'`. It is generally recommeneded to use just one variety or the other for string literals in a given project (mixing the two is seen as bad form).

The `str()` function returns a string representation of the object it is given as input. For instance, `str(4)` returns `"4"` and `str(True)` returns `"True"`. This is especially helpful when joining strings as in the following example:

```
x = 3.14159
print("x = " + str(x) + " m")
```

A convenient way to construct nice strings is the **formatted string (f-string)** literals. A simple f-string that has the same value as what is printed in the example above is `f"x = {x} m"`. Executable expressions are inserted in braces `{}` within the f-string. Note that the `str()` function is automatically called, which makes for a nicer syntax.

A **format specifier** can also be applied to expressions in an f-string. These have the general form

```
:[[fill]align][sign][z][#][0][width][group][.prec][type]
```

Each of these terms is described in table 1.2 and table 1.3. In the example above, we could format the printing of `x` in fixed-point format (`f`) with a precision (`.prec`) of 3 decimal places with

```
print(f"x = {x:.3f} m")      # => x = 3.142
```

In the following example, we use scientific notation (`e`) with precision (`.prec`) of 4:

```
x = 0.00123
print(f"x = {x:.4e} m")      # => x = 1.2300e-03 m
```

Note that the number of **significant digits** is 1 greater than the precision in this format. In the following example, we use binary (`b`) with 0-padding:

```
x = 3
print(f"x = {x:04b} (in binary)")       # => x = 0011 (in binary)
```

Table 1.2. Format specifier terms.

Term	Values (if any)	Default	Effect
`:`			Separates the format specifier from the expression
`fill`	Any character	space	Character to pad with when value doesn't use the entire field width
`align`	< (left) | > (right) | ^ (center) | =	<	How to justify when value doesn't occupy the entire field width
`sign`	+ (explicit +) | − (no plus) | space (space but no plus)	−	How a sign appears for numeric values
`z`	`z`		Coerces `-0.0` to `0.0`
`#`	`#`		Use the alternate output form for numeric values
`0`	`0`		Pad on the left with zeros instead of spaces
`width`	Positive integers		Minimum width of (number of characters in) the field
`group`	`,` | `_`		Grouping character (thousands separator) for numeric output

Term	Values (if any)	Default	Effect
`.prec`	Nonnegative integers	varies with `type`	Digits after the decimal point for floating-points, maximum width for strings
`type`	See table 1.3	`s` (strings) or `d` (numbers)	Specifies the presentation type, which is the type of conversion performed on the corresponding argument

Table 1.3. Format specifier types.

Input Class	Format Type	Meaning
String	`s`	String
String	None	String (same as `s`)
Integer	`b`	Binary
Integer	`c`	Character
Integer	`d`	Decimal integer
Integer	`o`	Octal
Integer	`x` or `X`	Hex (lowercase or uppercase)
Integer	`n`	Local decimal number (similar to `d`)
Integer	None	Same as `d`
Floating-point	`e` or `E`	Scientific notation (lowercase or uppercase)
Floating-point	`f` or `F`	Fixed-point notation (lowercase or uppercase)
Floating-point	`g` or `G`	General format (lowercase or uppercase)
Floating-point	`n`	Local general format
Floating-point	`%`	Percentage
Floating-point	None	Same as `g`

The `str` class has several methods. We have already seen the `capitalize()` method applied in section 2.3.2. Table 1.4 describes several frequently used string methods.

Table 1.4. Some particularly useful string methods.

Method	Description
`capitalize()`	Convert the first character to uppercase
`count()`	Return the count of the specified value occurrences
`endswith()`	If the string ends with the specified value, return `True`
`find()`	Return the position where the specified value is found
`index()`	Return the position where the specified value is found
`isalpha()`	If all characters are alphabetic, return `True`
`isdecimal()`	If all characters are decimals, return `True`
`isdigit()`	If all characters are digits, return `True`
`isnumeric()`	If all characters are numeric, return `True`
`join()`	Convert iterable elements into a single string
`lower()`	Convert the string to lowercase
`replace()`	Return a string with the specified value replaced
`rindex()`	Return the last position where the specified value is found
`rsplit()`	Split at the specified separator, return a list
`split()`	Split at the specified separator, return a list
`splitlines()`	Split at line breaks, return a list

Method	Description
`startswith()`	If the string starts with the specified value, return `True`
`strip()`	Return a trimmed version of the string

1.3.3 Iterable Objects and Dictionaries

In Python, an **iterable object** is one that contains a collection of **elements** and defines, for each element, which element is next. In the following sections, we will consider some built-in iterable classes (types).

Box 1.1 Further Reading

- Python Community (2024a; § The Python Tutorial: 9 Classes), on classes, objects, and methods
- Python Community (2024a; § Python Standard Library: Built-in Types), on the basic built-in types

1.4 Lists

KX

The **list** class defines an ordered set of elements. These elements can be of any class, and do not need to match within a list. Lists can be nested to create a list of lists. The basic syntax for creating a list of elements e*x* is `[e1, e2, ..., en]`. Consider the following list assignments:

```
int_list = [3, 9, 3, -4, 0]          # Duplication allowed
str_list = ["foo", "bar", "baz"]
com_list = [int_list, str_list]      # List of lists
mix_list = [8.41, "foo", [7]]        # Mixing element types
```

1.4.1 Accessing List Elements

Because the elements of a list have an order, they can be referred to via an **index**, a mapping of integers to elements. In Python, the first element in the list has index `0` and subsequent elements have indices of increasing values, `1`, `2`, `3`, and so on. The syntax for accessing the element with index `i` of a list `l` is `l[i]`. For instance, elements from the previously defined lists can be accessed as follows:

```
int_list[0]                          # => 3
int_list[3]                          # => -4
str_list[2]                          # => "baz"
mix_list[2]                          # => [7]
```

Negative indices are used to access elements from the end of a list. For instance, for `int_list` above,

```
int_list[-1]                         # => 0
int_list[-2]                         # => -4
```

This is particularly useful when we want to access the last element of a list, which we see has index `-1`.

A selection of elements from a list can be accessed via **slicing**, which has the syntax `l[start:stop]` or `l[start:stop:step]`. For instance,

```
l = [0, 1, 2, 3, 4]
l[0:3]                               # => [0, 1, 2]
l[2:4]                               # => [2, 3]
l[0:-1]                              # => [0, 1, 2, 3] (no last item!)
l[0:]                                # => [0, 1, 2, 3, 4]
l[0::2]                              # => [0, 2, 4] (every two elements)
```

It is important to note that the slice does not include the `stop` index; rather, the slice's last value is from index `stop-1`. As we see in the third slice example, this means the normal syntax for slicing through the final element (i.e., the element with index `-1`) does not include that element. To include the final element, leave off an index for `stop`, as shown in the fourth and fifth examples.

1.4.2 Mutability

Lists are **mutable**; that is, they can be mutated (changed). This is unlike most built-in types, which are **immutable** and cannot be changed. The mutability for frequently used built-in types is shown in table 1.5.

Table 1.5. Mutability of commonly used built-in types.

Data Type	Built-in Class	Mutability
Numbers	`int`, `float`, `complex`	Immutable
Strings	`str`	Immutable
Tuples	`tuple`	Immutable
Booleans	`bool`	Immutable
Lists	`list`	Mutable
Dictionaries	`dict`	Mutable
Sets	`set`	Mutable

The mutability of lists allows us to change their elements. The syntax for assigning a new value `v` to an element with index `i` of a list `l` is `l[i] = v`. For instance,

```
l = ["Hello", "World", "!"]
l[1] = "Stranger"
print(l)
```

returns

```
['Hello', 'Stranger', '!']
```

Note that although strings are immutable, a list of strings is mutable. This means `"Stranger"` is not at the same location in memory as was `"World"`.

1.4.3 Methods

Lists have several methods for mutating themselves, which are given in table 1.6.

Table 1.6. Commonly used list methods for a list `l`.

Method	Description
`l.append(item)`	Append `item` to the end of `l`
`l.clear()`	Remove all items from `l`
`l.extend(iterable)`	Concatenate `l` with the contents of `iterable`
`l.index(x[, start[, end]])`	Return the index of the first instance of `x` in `l[start:end]`
`l.insert(index, item)`	Insert `item` into `l` at `index`
`l.pop(index)`	Return and remove the item at `index`
`l.pop()`	Return and remove the last item
`l.remove(item)`	Remove `item`'s first occurrence
`l.reverse()`	Reverse the items of `l`
`l.sort(key=None, reverse=False)`	Sort the items of `l`

For example, an element can be inserted into a list as follows:

```
l = ["zero", "one", "three"]
l.insert(2, "two")
print(l)
```

which returns

```
['zero', 'one', 'two', 'three']
```

When using most list methods, we often do not assign the returned value from the expression. This is because most of these expressions return a value of `None`. For instance, from the previous example,

```
print(l.insert(2, "two"))
```

returns

```
None
```

Such methods are simply operating on the original list object and do not return that object. This is a common idiom in Python programming, and many mutable classes behave similarly.

Example 1.3

Write a program that removes the second occurrence of the element 3 from the following list:

```
l = [1, 2, 3, 0, 3, 4, 3]
```

The `remove()` method might seem promising, but it only removes the first occurrence of the element. Instead, let's identify the index of the second occurrence. The `index(x[, start[, end]])` method allows us to identify the index of the first occurrence or the first occurrence between `start` and `end`. So our strategy is to find the index `i_first` of the first occurrence with `index()`, then narrow our search to the rest of the list after `i_first` to the end of the list, identifying the second index `i_second`. Finally, we can remove the element at `i_second` with the pop method.

The following program implements this strategy.

```
l = [1, 2, 3, 0, 3, 4, 3]
x = 3                                # element we are removing
i_first = l.index(x)                 # first occurrence index
i_second = l.index(x, i_first+1)     # second occurrence index
l.pop(i_second)                      # removes second occurrence
print(f"l without second {x}: {l}")
```

This prints

```
l without second 3: [1, 2, 3, 0, 4, 3]
```

1.5 Tuples and Ranges

Python has a built-in **tuple** class `tuple` is very similar to a `list` in that it is an ordered collection of elements. The term "tuple" is a generalization of the terms "single," "double," "triple," "quadruple," and so on. The primary difference between a tuple and a list is that a tuple is immutable, so its elements can't be changed. The syntax for a tuple literal of elements e*x* is `(e1, e2, ..., en)`. The elements can each be of any type, including tuples. For example, the following statements return tuples:

```
(0, 1, 2, 4, 5)
("foo", "bar", "baz")
([0, 1], [2, 3])
((0, 1), (2, 3))
(0, "foo", [1, 2], (3, 4))
```

Elements of a tuple can be accessed via the same syntax as is used for lists, including slicing. For instance,

```
t = (0, 1, 2)
t[1]              # => 1
t[0:2]            # => (0, 1)
t[1:]             # => (1, 2)
```

Because tuples are immutable, there are only two built-in tuple methods, `count()` and `index()`. The `count()` method returns the number of times its argument occurs in the tuple. For instance,

```
t = (-7, 0, 7, -7, 0, 0)
t.count(-7)      # => 2
```

The `index()` method returns the index of the first occurrence of its argument. For instance,

```
t = ("foo", "bar", "baz", "foo", "bar", "baz", "baz")
t.index("baz")  # => 2
```

The **range** built-in type is a compact way or representing sequences of integers. A `range` can be constructed with the `range(start, stop, step)` constructor function, as in the following examples:

```
list(range(0, 3, 1))     # => [0, 1, 2]
list(range(2, 6, 1))     # => [2, 3, 4, 5]
list(range(0, 3))        # => [0, 1, 2] (step=1 by default)
list(range(3))           # => [0, 1, 2] (start=0 by default)
```

Note that we have wrapped the ranges in `list()` functions, which converted each range to a list. This was only so we can see the values it represents; alone, an expression like `range(0, 3)` returns itself. This is why a range is such a compact data point—all that needs to be stored in memory are the `start`, `stop`, and `step` arguments because the intermediate values are implicit.

1.6 Dictionaries

The built-in Python **dictionary** class `dict` is an unordered collection of elements, each of which has a unique **key** and a **value**. A key can be any immutable object, but a string is most common. A value can be any object. The basic syntax to create a `dict` object with keys `k`*x* and values `v`*x* is `{k1: v1, k2: v2, ...}`. For instance, we can define a `dict` as follows:

```
d = {"foo": 5, "bar": 1, "baz": -3}
```

Accessing a value requires its key. To access a value in dictionary `d` with key `k`, use the syntax `d[k]`. For example,

```
d = { # It is often useful to break lines at each key-value pair
    "name": "Spiff",
    "age": 33,
    "occupation": "spaceman",
    "enemies": ["Zorgs", "Zargs", "Zogs"]
}
print(f"{d['name']} is a {d['age']} year old"
      f"{d['occupation']} who fights {d['enemies'][0]}.")
```

This returns

```
Spiff is a 33 year old spaceman who fights Zorgs.
```

A value `v` with key `k` can be added to an existing dictionary `d` with the syntax `d[k] = v`. For instance, (Filik et al. 2019)

```
d = {} # Empty dictionary
d["irony"] = "The use of a word to mean its opposite."
d["sarcasm"] = "Irony intended to criticize."
```

Dictionaries are mutable; therefore, we can change their contents, as in the following example:

```
d = {}
d["age"] = 33   # d is {"age": 33}
d["age"] = 31   # d is {"age": 31}
```

Dictionaries have several handy methods; these are listed in table 1.7.

Table 1.7. Dictionary instance methods for dictionary instance `d` and class method for class `dict`.

Methods	Descriptions
`d.clear()`	Clears all items from `d`
`d.copy()`	Returns a shallow copy of `d`
`dict.fromkeys(s[, v])`	Returns a new `dict` with keys from sequence `s`, each with optional value `v`
`d.get(k)`	Returns the value for key `k` in `d`
`d.items()`	Returns a view object of key-value pairs in `d`
`d.keys()`	Returns a view object of keys in `d`
`d.pop(k)`	Removes and returns the value for key `k` in `d`
`d.popitem()`	Removes and returns the last-inserted key-value pair from `d`
`d.setdefault(k, v)`	Returns the value for the key `k` in `d`; inserts `v` if absent
`d.update(d_)`	Updates `d` with key-value pairs from another dictionary `d_`
`d.values()`	Returns a view object of values in `d`

Note that most of these methods apply to dictionary instance `d`, either mutating `d` or returning something from `d`. However, the `fromkeys()` method is called from the class `dict` because it has nothing to do with an instance. Such methods are

called **class methods**; the other methods we've considered thus far are **instance methods**.

Dictionary **view objects**—returned by `items()`, `keys()`, and `values()`—are dynamically updating objects that change with their dictionary. For instance,

```
d = {"a": 1, "b": 2}
d_keys = d.keys()
print(f"View object before: {d_keys}")
d["c"] = 3
print(f"View object after: {d_keys}")
```

This returns

```
View object before: dict_keys(['a', 'b'])
View object after: dict_keys(['a', 'b', 'c'])
```

View objects can be converted to lists with the `list()` function, as in `list(d_keys)`.

Example 1.4

Write a program that meets the following requirements:

1. It defines a list of strings `names = ["Mo", "Jo", "Flo"]`
2. It constructs a `dict` instance `data` with keys from the list `names`
3. It creates and populates a sub-`dict` with the follow properties for each name:
   a. Mo—year: sophomore, major: Mechanical Engineering, GPA: 3.44
   b. Jo—year: junior, major: Computer Science, GPA: 3.96
   c. Flo—year: sophomore, major: Philosophy, GPA: 3.12
4. It prints each of the students' name and year
5. It replaces Jo's GPA with 3.98 and prints this new value
6. It removes the entry for Mo and prints a list of remaining keys in `data`

The following program meets the given requirements:

```
names = ["Mo", "Jo", "Flo"]
data = dict.fromkeys(names) # => {"Mo": None, "Jo": None, "Flo": None}

#%% Populate Data
data["Mo"] = {}
data["Mo"]["year"] = "sophomore"
data["Mo"]["major"] = "Mechanical Engineering"
data["Mo"]["GPA"] = 3.44
data["Jo"] = {}
data["Jo"]["year"] = "junior"
data["Jo"]["major"] = "Computer Science"
data["Jo"]["GPA"] = 3.96
data["Flo"] = {}
data["Flo"]["year"] = "sophomore"
data["Flo"]["major"] = "Philosophy"
data["Flo"]["GPA"] = 3.12

#%% Data Operations and Printing
print(f"Mo is a {data['Mo']['year']}. "
      f"Jo is a {data['Jo']['year']}. "
      f"Flo is a {data['Flo']['year']}.")
data["Jo"]["GPA"] = 3.98
print(f"Jo's new GPA is {data['Jo']['GPA']}")
data.pop("Mo")
print(f"Names sans Mo: {list(data.keys())}")
```

This prints the following in the console:

```
Mo is a sophomore. Jo is a junior. Flo is a sophomore.
Jo's new GPA is 3.98
Names sans Mo: ['Jo', 'Flo']
```

1.7 Functions

In Python, **functions** are reusable blocks of code that accept input arguments and return one or more values. As we have seen, a method is a special type of function that is contained within an object. We typically do not refer to methods as "functions," instead reserving the term for functions that are not methods. A function that computes the square root of the sum of the squares of two arguments can be defined as:

```
def root_sum_squared(arg1, arg2):
    sum_squared = arg1**2 + arg2**2
    return sum_squared**(1/2)
```

The syntax requires the block of code following the `def` line to be indented. A block ends where the indent ends. The indent should, by convention, be 4 space characters. The function ends with a **return statement**, which begins with the keyword `return` followed by an expression, the value of which is returned to the caller code. The variable `sum_squared` is created inside the function, so it is local to the function and cannot be accessed from outside. **Calling** (using) this function could look like

```
root_sum_squared(3, 4)
```

This call returns the value `5.0`.

The arguments `arg1` and `arg2` in the previous example are called **positional arguments** because they are identified in the function call by their position; that is, `3` is identified as `arg1` and `4` is identified as `arg2` based on their positions in the argument list. There is another type of argument, called a **keyword argument** (sometimes called a "named" argument), that can follow positional arguments and have the syntax <key>=<value>. For instance, we could augment the previous function as follows:

```
def root_sum_squared(arg1, arg2, pre="RSS ="):
    sum_squared = arg1**2 + arg2**2
    rss = sum_squared**(1/2)
    print(pre, rss)
    return rss
```

The `pre` positional argument is given a default value of `"RSS ="`, and the function now prints the root sum square with `pre` prepended. Calling this function with

```
sum_squared(4, 6)
```

prints the following to the console:

```
RSS = 7.211102550927978
```

Alternatively, we could pass a value to `pre` with the call

```
sum_squared(4, 6, pre="Root sum square =")
```

which prints

```
Root sum square = 7.211102550927978
```

1.8 Branching

CL

There are special statements in all programming languages that allow the programmer to control which portions are to be executed next (or at all); that is, the **control flow**. The primary forms of control flow statements are **branching** and **looping**, and we introduce branching in this section and looping in section 1.9.

1.8.1 Branching with if/elif/else Statements

Branching control flow statements are based on logical conditions that are tested by the statement. The primary branching statements in Python are the `if`/`elif`/`else` statements. For instance, consider the following statements:

```
if x < 0:
    print("negative")
elif x == 0:
    print("zero")
else:
    print("positive")
```

If `x` is less than 0, it will print `negative`; if `x` is equal to 0, it will print `zero`, and otherwise (when `x` is positive) it will print `positive`. Note that the blocks of code that follow the branching statements must be indented. The `elif` (i.e., else if) and `else` statements are optional, and there can be multiple `elif` statements. Once a condition is met and the corresponding block executed, the rest of the control statements in the block are skipped.

The conditional expression is evaluated to a `bool` type (class). A `bool`ean object can have one of two possible values, `True` and `False`. If the conditional expression of a branching statement evaluates to `True`, its corresponding block of code is executed. Note that Python will evaluate non-`bool`ean conditional expression value with the built-in `bool()` function. For instance, if the conditional expression evaluates to a string `"foo"`, it will be evaluated as `bool("foo")`, which, like all nonempty strings, evaluates to `True`. However, an empty string `""` evaluates to `False`.

Example 1.5

Write and test a Python program that prints a string variable if it is nonempty, and prints `Empty string` otherwise.

We will want to test our program on a nonempty and an empty string, so we will want to reuse our code; this indicates the use of a function definition. Consider the following program:

```
def print_nonempty(s):
    if s:
        print(s)
    else:
        print("Empty string")

print_nonempty("This should print")
print_nonempty("")      # This should print "Empty string"
```

The `if` statement has conditional expression `s`, which should be a string. Therefore, if it is nonempty, `print(s)` will evaluate. Otherwise (i.e., if `s` is an empty string), the statement `print("Empty string")` will evaluate. As we expect, the program prints the following to the console:

```
This should print
Empty string
```

1.8.2 Branching with `match`/`case` Statements

In Python 3.10, a new kind of branching statement was introduced: `match`/`case`. Its use is never strictly necessary, but it can make a program more readable. For example,

```
if s == "red":
    print("red")
elif s == "blue":
    print("blue")
else:
    print("other")
```

can be written alternatively as

```
match s:
    case "red":
        print("red")
    case "blue":
        print("blue")
    case _:
        print("other")
```

In the third case _ matches when there is no other match. Once there is a match, no other cases are tested. If there is no match (and _ is not given as a case), none of the code blocks are evaluated.

There are more advanced uses of `match`/`case` statements in which patterns can be matched. See Python Community (2024a; § 4.6) for more details.

1.8.3 Branching with `try`/`except`/`finally` Statements

Sometimes a statement can yield an **exception**, which is not a syntax error, but has a similar effect in that it can stop the execution of the program. Common exceptions include `ZeroDivisionError`, `NameError` and `TypeError`.

In general, an exception stops the execution of a program; however, certain exceptions can be anticipated and dealt with accordingly, which is called exception handling. One of the primary ways to handle exceptions is to use the `try`/`except`/`finally` statements. We can think of these statements as branching statements that branch based on exceptions. For instance consider the following function definition:

```
def plus_7(x):
    try:
        y = x + 7
    except:
        y = x
    return y
```

If we can add `7` to `x`, which is the case when `x` is a number, the `try` statement will execute, the `except` statement will be skipped, and the sum will be returned. If, however, we cannot add `7` to `x`, which is the case when `x` is nonnumeric, the `try` statement will raise an exception, so the `except` statement will be executed; this returns the input without change.

We will later return to exception handling to consider more advanced usage, including the `finally` statement.

1.9 Looping

Repeating blocks of code by calling a function more than once, as in example 1.5, can get cumbersome when it needs to be repeated many times. A **loop** repeats a block until some stopping condition is met. One type of loop in Python is a `while` loop, which repeats a block of code while its conditional expression evaluates to `True`. For instance,

```
n = 0           # Initialize n
while n < 5:
    print(n)
    n += 1      # Increment n (i.e., n = n + 1)
```

The loop evaluates the conditional expression `n < 5` and, if in fact `n < 5`, executes the block of code. After the block finishes, the test is repeated and potentially the block of code. This will repeat indefinitely, until the conditional expression evaluates to `False`, in which case the loop exits and execution resumes after the code block. The block will be executed 5 times, printing `0` through `4` to the console.

Another type of Python loop is a `for` loop, which has no explicit conditional expression, instead iterating through an iterable object like a list, , until it reaches the end. For example,

```
l = ["foo", "bar", "baz"]
for s in l:
    print(f"Say {s}")
```

This prints

```
Say foo
Say bar
Say baz
```

It is common to loop through a `range` with a `for` loop, as in the following:

```
for k in range(2, 8):
    print(k, end=" ") # Prints on the same line
```

This prints the following to the console:

```
2 3 4 5 6 7
```

Often, a loop index is required inside a `for` loop. The syntax for this requires an identifier for the index and an `enumerate` type object to be iterated through. The constructor function `enumerate()` assigns an index to each element of its iterable argument (e.g., a list). For instance,

```
names = ["Manny", "Bella", "Amadeus"]
signs = ["Libra", "Virgo", "Sagittarius"]
for i, name in enumerate(names):
    print(f"{name} is a {signs[i]}")
```

This prints the following to the console:

```
Manny is a Libra
Bella is a Virgo
Amadeus is a Sagittarius
```

Looping through a dictionary is similar, but we need the `items()` of the dictionary for the key-value pair, as follows:

```
sounds = {"dog": "woof", "cat": "meow", "fox": "ring-ding-ding"}
for k, v in sounds.items():
    print(f"The {k} says '{v}'")
```

This prints the following to the console:

```
The dog says 'woof'
The cat says 'meow'
The fox says 'ring-ding-ding'
```

1.10 Problems

Problem 1.1 BA Write a program with the following requirements:

a. It defines variables for the following quantities:

$$x = 5.2 + j3.4, \quad y = -17, \text{ and } \quad z = 0.02,$$

where j is the imaginary number $\sqrt{-1}$.

b. It computes and prints the following quantities:

$$x + y, \quad xyz, \text{ and } \quad 4x^3 - 8xy + 6y^2.$$

c. It further computes and prints the following quantities:

$$|x|, \quad \overline{xy}, \text{ and } \quad \Re(x),$$

where $|\cdot|$ is the absolute value, $\overline{\cdot}$ is the complex conjugate, and $\Re(\cdot)$ is the real part.

Problem 1.2 SB Write a program with the following requirements:

a. It defines a variable for a list with the following elements:

```
4, -12, 6, -14, 8, -16
```

b. It prints the first and last elements of the list
c. Using list slicing, it prints the first three elements of the list
d. Using list slicing, it prints the last three elements of the list
e. Using list slicing, it prints every other element, starting with the first element
f. It computes and prints the length of the list (consider using the built-in function `len()`)
g. It computes and prints the sum of the list elements (consider using the built-in function `sum()`)

Problem 1.3 I2 Write a program with the following requirements:

a. It defines a variable for a list with the following elements:

```
32, 41, 58, 34, 24, 53, 46, 41
```

b. It computes and prints the mean of the list items (consider using the built-in `sum()` and `len()` functions)
c. It finds and prints the maximum and minimum values in the list (consider using the built-in `max()` and `min()` functions)

d. It finds and prints the indices of the maximum and minimum values in the list (consider using the `index()` method)
e. It sorts and prints the sorted list (minimum to maximum; consider using the `sort()` method)

Problem 1.4 VZ Write a program with the following requirements:

a. It defines a function `which_number()` that takes a single argument and, if it is an `int`, `float`, or `complex` object, returns the strings `"int"`, `"float"`, or `"complex"`. If the argument is not a number, it returns `None`.
b. It tests the function and prints its return value on the following inputs:
   i. `42`
   ii. `3.92`
   iii. `complex(2, -3)`
   iv. `"3.92"`
   v. `[2, 0]`

Problem 1.5 X4 Write a function `capital_only(l)` with the following requirements:

a. It accepts as input a list `l`
b. It checks that all elements are strings; it raises an exception, otherwise, with

```
raise ValueError(
    "All elements must be strings"
)
```

c. It returns a list (not the same list[2]) with only the strings that begin with a capital letter
d. It returns the proper output for the following inputs (demonstrate this in the program):
   i. `["Foo", "Bar", "Baz"]`
   ii. `["Foo", "bar", "Baz"]`
   iii. `["Foo", 0, 1, "Bar", 2]`

Problem 1.6 X0 Write a program with the following requirements:

a. It defines a function `float_list()` that takes a single `list` argument and returns a new list with all elements converted to `float`s
b. If the input is not a `list`, it returns an empty list
c. If an element is an `int`, it should be converted to a `float`
d. If an element is a `str`ing, the program should attempt to convert it to a `float`

2. Because a list is mutable, we must take care not to mutate a list inside a function (except in rare cases when this behavior is desired).

   i. For strings like `"3.24"`, the `float()` function will work
   ii. For strings like `"foo"`, the `float()` function will throw a `ValueError`; consider using `try` and `except` statements

e. If an element cannot be converted to a `float`, it should be left out of the returned list
f. If an element is `complex`, it should remain so
g. Test and print the returned list for the following inputs:
   i. `[1.1, 0.2, 4.2, -30.2]`
   ii. `[3, 42, -32, 0, 3]`
   iii. `[1-3j, 2, 0.3]`
   iv. `["1.2", "8", "-3.9"]`
   v. `["0.4", "dog", None, 8]`
   vi. `3.4`

2 The Structure, Style, and Design of Programs

With the development environment and basic elements of a Python program described in chapter 1, we can write a great many interesting programs. In this chapter, we consider how these programs should be structured and styled.

2.1 Python Interpreters and Interactive Sessions

RZ

When we execute (i.e., run) a Python program, an **interpreter** translates the program code into an efficient intermediate representation and carries out the corresponding instructions, which are ultimately represented in the lowest-level computer language called **machine code**. Instructions in machine code can be given directly to the processor and thereby executed.

An interpreter is a program that translates another program line-by-line. There is another way of translating a program (written in a programming language) to machine code—compiling. A compiler takes the entire program at once and translates it into a highly optimized machine code program, ready for execution. An interpreter cannot optimize a program as much as can a compiler, but there are advantages to using an interpreter, including that programs can be run interactively.

The official Python interpreter CPython was installed to our development environment in section 1.2. The basic way to run a Python program `hello.py` is in a terminal window with the command

```
python hello.py
```

Here the interpreter program `python` is called to interpret `hello.py` and the results are printed to the terminal. Programs written in a file like `hello.py` are called **scripts**.

Another way to run a Python script is within an **interactive session** (i.e., interactive shell, read-evaluate-print loop (REPL), or kernel) that runs lines of code as they are entered by the programmer. There are multiple programs that provide interactive Python sessions, including the standard one provided by the CPython distribution and invoked in a termal window with the command `python` (without

a script given). This command causes the terminal to start an interactive session with a prompt that appears similar to the following:

```
$ python
Python 3.<specific version number> | <additional information>
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Python statements can be entered here and executed by pressing the key ⏎. For instance, a list can be created and sorted:

```
>>> l = [1.3, 0.8, 6.1, 3.9]
>>> l.sort()
>>> l
[0.8, 1.3, 3.9, 6.1]
```

Note that no explicit `print()` function call is necessary for objects to be printed in the interactive session. When a statement returns a value of probable interest, the session automatically prints it.

A more richly featured program for interactive Python sessions is **IPython**. The Spyder IDE in our development environment provides an IPython console (like a terminal) in the lower-right corner in the default layout.

```
In[1]: x = 4.29
In[2]: y = -38.1
In[3]: x**2 + x*y + y**2
Out[3]: 1306.5651
```

Within an IPython interactive session, a script can be run with the `runfile()` function. This is precisely what Spyder does when a script in the editor is run via the Run › Run menu item, the ▶ button, or the F5 key. An advantage to this technique is that the variables defined in the script now enter the existing IPython interactive session, which allows us to play with them for further analytic exploration or for debugging purposes. Note however that the reverse is not the case: variables from the interactive session are not available to the script run within that session. This allows us to have confidence that a script developed with the use of `runfile()` in an interactive session can be run outside that session. Therefore, in this book, we need not be concerned about which method—direct calling of the interpreter via `python` or the use of `runfile()` within an interactive session—was used to run a Python script.

2.2 Scripts, Modules, and Imports

As we have seen, the main code for a Python program is written in a file called a script. There are a few reasons we often want to use multiple files for a single program:

1. For large programs, a single file becomes unwieldy or difficult to navigate.
2. Portions of our program, like function and class definitions, are self-contained and potentially useful for other programs.
3. A code library (with its own files) is available to perform certain tasks without writing that code ourselves (see section 2.3).

In these cases, files containing definitions and statements (usually constants, functions, and classes) called **modules** can be **imported** to a main script. We often write our own modules for the first two cases above; that is, when our script gets long or certain definitions may be useful for other programs. For example, if we have defined a function `do_something()` in a module file `a_module.py` placed in the same directory as our main script, we can import the module and use that function in the main script with the following statements:

```
import a_module          # Import the module
a_module.do_something() # Call the imported function
```

Note that the function is available as an attribute of `a_module` (sans `.py`); that is, to access the function `do_something()`, we must call `a_module.do_something()`. This keeps us from accidentally overwriting names in the main main script or from other modules. Occasionally, we may want a specific definition from a module to be directly available in the script. This can be achieved with the following statements:

```
from a_module import do_something # Import function from the module
do_something()                    # Call the imported function
```

Occasionally, the name of a module is longer than is convenient to use within a script. In this case, we can give the module a nickname, as in

```
import a_module as am    # Import the module
am.do_something()        # Call the imported function
```

For a project with several modules, it is best to move modules into subdirectories with names that clearly indicate their functionality. For instance, with a module in one subdirectory `blue_things/cyanotype.py` and another module in another subdirectory `red_things/redscale.py`, importing these modules requires the following statements:

```
import blue_things.cyanotype
import red_things.redscale
```

Note that the dot "`.`" indicates that a directory contains the module that follows.

2.3 The Python Standard Library and Packages

This section introduces the importing of modules from the Python standard library and the importing of external code libraries (packages).

2.3.1 The Standard Library

Like many programming languages, Python has an extensive **standard library** with many built-in data types, constants, functions, and modules included. We have encountered some of these already, and this section gives a very short introduction to some additional aspects of the library.

Some of the standard library is available in the built-in namespace, such as the constants `True`, `False`, and `None` and the functions `print()`, `len()`, and `type()`. However, much of the standard library requires the `import`ing of modules. A list of modules of particular interest to the engineer is given in table 2.1.

Table 2.1. Python standard library modules of particular interest to the engineer.

Module	Description
`math`	Math constants and functions for integers and real numbers.
`cmath`	Math constants and functions for integers, real numbers, and complex numbers.
`random`	Functions for generating pseudorandom numbers.
`os`	Functions for interacting with the computer's operating system and file system.
`pathlib`	Classes for representing file paths in an operating-system independent way.
`json`	Functions for importing and exporting data in the universal JSON format.
`pickle`	Functions for saving and loading objects to files in serialized (compact) form.

Just as with our own modules, we can `import` a module from the standard library with

```
import math
```

and the related variations of `import`. The standard library modules are always in the Python **search path**, which is a list of directories in which Python searches for modules. The search path begins locally, so if you create a module `math.py`, the search path will find it before the standard library version.

2.3.2 Packages

In addition to the standard library modules, a vast collection of Python **packages** can be installed and `import`ed. A package is a collection of modules. Packages are created to organize code into reusable units and distribute them to others.

The official source for Python packages is the Python Package Index (PyPI) (Python Community 2024b). The `pip` program distributed with Python is the most popular tool for installing and managing packages. Packages can be installed in Anaconda environments with `pip`, but the use of its own package manager called `conda` is preferred. The `base` Anaconda environment comes with many preinstalled packages useful for engineering computing. The installation process for installing a package includes adding the package to the Python path so that it is available to all your Python programs.

Once a package is installed, it can be `import`ed in a script. Most packages import by default one or more modules; this allows us to `import` the package in our script without individually importing each module. For instance, if we would like to use a function `do_something()` in the `foo.py` module of the `pkg` package, we could write the following:

```
import pkg                # Import the entire package
pkg.foo.do_something()  # Call a function in a module loaded by default
```

If the module is not loaded by default, or if we would only like to load a specific module, we can manually `import` the module in the usual way:

```
import pkg.foo            # Import the module
pkg.foo.do_something()  # Call a function in the module
```

Often, packages will import some important functions into its top-level namespace such that they can be called with a shorter name. In the example above, the package could elevate `do_something()` to its top-level namespace such that it can be called via `pkg.do_something()`.

Packages can contain packages, called **subpackages**. Simple packages do not require this nesting feature, but large and complex packages may.

You may one day create a package of your own. All that is required is to place your modules into a directory. If you place a special file named `__init__.py` in the directory `my_pkg`, it will be executed whenever the package is loaded.[1] Often, we want to load certain (or all) modules in this file such that they are imported by default when the package is loaded.

Your package can be distributed via PyPI or another means.

1. For earlier versions of Python, the `__init__.py` file was obligatory for a package. Now it is optional but advisable.

Box 2.1 Further Reading

- Python Community (2024a; § The Python Standard Library)
- Python Community (2024a; § The Python Tutorial, 10. Brief Tour of the Standard Library)
- Python Community (2024b), to browse PyPI packages
- Python Community (2024c), for information about creating and distributing packages

2.4 Namespaces, Scopes, and Contexts

A **namespace** is a binding of (i.e., a map from) names (identifiers) to objects. Each name is unique within a namespace. For instance, there can be only one variable `x`. In Python, as in many programming languages, namespaces are created and destroyed throughout the execution of a program. When a main script is run, the Python interpreter creates (and never destroys) the **built-in namespace** that includes mappings for several built-in objects such as the functions `print()`, `len()`, and `abs()` and the constants `True`, `False`, and `None`.

As we saw in section 2.2, the names in a namespace for an imported module `a_module` begin with the name of the module, as in `a_module.do_something()`. Or, if the module was imported with an alias, as in `import a_module as am`, the names in its namespace begin with `am`.

The main script or a module has a top-level namespace called the **global namespace**. Names defined in the script or module and outside of any function or class definition go into this top-level namespace. The execution of a function or class creates a new namespace for it. This is true for nested function and class definitions, as well. Therefore, a hierarchy of namespaces is created with nested function and class definitions. At the bottom of this hierarchy is an innermost **local namespace**. Levels below the global namespace and above a local namespace are called **non-local namespaces** (i.e., enclosing namespaces).

The **scope** of a name binding (to an object) is the portion of the code of a program in which the name is bound (i.e., where it can be used).[2] The scope of `x = 3` is the part of the code in which the use of `x` will return that `3`. The **context** for a given portion of a program is the collection of all bound names and the ordering of namespaces searched when a name is used. The context of a scope of names in a local namespace has the following search priority:

1. Local namespace

2. Sometimes the term "scope" is used to mean what we call a "context of a scope." We will try to avoid this usage, but it is quite common.

2. Non-local namespaces
3. Global namespace
4. Built-in namespace

For instance, consider the namespaces, scopes, and contexts for the following script:

```
x = 3
print(f"Global x: {x}")
def plus_7(y):
    x = y + 7
    print(f"Local x: {x}")
    return x
plus_7(x)
print(f"Global x: {x}")
```

This prints the following to the console:

```
Global x: 3
Local x: 10
Global x: 3
```

To interpret these results, we see that the statement `x = 3` binds the name `x` in the global namespace such that `Global x: 3` is printed. The context for this portion of code is the collection of bindings for the names in the global and built-in namespaces and the search priority (1) global namespace and (2) built-in namespace. The `plus_7()` function definition creates a new local namespace in which `x` is bound with the assignment `x = y + 7`. The context for the function code block is the set of bindings for the names in the local, global, and built-in namespaces and the search priority (1) local namespace, (2) global namespace, and (3) built-in namespace. Therefore, the use of `x` here searches the local namespace first; finding one upon the function being called, it prints (in this case) `Local x: 10`. Finally, we see that the global namespace `x` has been unchanged by the local assignment.

Example 2.1

In the previous example, if we remove the local assignment `x = y + 7`, what happens?

Because there is no binding of `x` in the local namespace of the function, `x` is not found here. Therefore, the global namespace is searched; the global namespace `x` is found and used within the function. This results in the program printing

```
Global x: 3
Local x: 3
Global x: 3
```

Note that if x had not been found in the global namespace, the built-in namespace would have been searched. Because this namespace also lacks a binding for x, a `NameError` would be raised.

The use of global or non-local names within a function or class definition is generally discouraged. It is difficult to read and debug code that refers to names outside of its local namespace. We prefer to pass necessary objects through input arguments. Even worse than the use of global names within a function or class definition is their reassignment or their bound object's mutation. Rarely necessary and nearly always a bad idea, this can be achieved with the use of the `global` and `nonlocal` keywords. Without these keywords, global and non-local names are read-only. With their use, global and non-local names can be reassigned and bound objects mutated, as in the following example:

```
x = 3
print(f"Global x: {x}")
def plus_7(y):
    global x
    x = y + 7
    print(f"Local x: {x}")
    return x
plus_7(x)
print(f"Global x: {x}")
```

This prints the following to the console:

```
Global x: 3
Local x: 10
Global x: 10
```

So we have altered x in the global namespace. Again, it is inadvisable to use this unless absolutely necessary.

2.5 Defining Classes

RF

Defining a custom class is an extremely useful way to use Python. A class object has two kinds of **class attributes**: **data attributes** that store data and **methods**, which, as we have already seen, are functions that belong to and often operate on instances of an object.

A custom class is a convenient way to represent many kinds of objects in engineering. Here are some examples with potential data attributes and methods included:

- A time-varying signal class with data attributes `periodic`, `period`, `amplitude`, and `frequency` and methods `rms()`, `abs()`, and `plot()`
- An experiment simulation class with data attributes `time`, `executed`, `input`, and `output` and methods `execute()`, `plot()`, and `save()`
- A truss class with data attributes `members`, `connections`, `pin_angles`, `member_forces`, and `reactions` and methods `analyze()`, `max_compression()`, `max_tension()`, and `max_reaction()`

The basic syntax for a class definition is as follows:

```
class ClassName:
    """Docsting description"""
    <Statement 1>
    <Statement 2>
    <etc.>
```

Data attributes can be defined via the usual variable assignment syntax and generally follow the docstring. Method definitions follow data attributes. Consider the following class definition to represent a screwdriver tool (perhaps in the context of a robot's inventory of available tools):

```
class Screwdriver:
    """Represents a screwdriver tool"""
    operates_on = "Screw"  # Class data attributes
    operated_by = "Hand"

    def drive(self, screw, angle):  # Method definition
        """Returns a screw object turned by the given angle"""
        return screw.turn(angle)
```

Any object that is an instance of the class `Screwdriver` will have the class attributes defined above. To create an instance (i.e., instantiate), call the class name as if it were a function with no arguments, as follows:

```python
sd1 = Screwdriver()  # Create an instance of the Screwdriver class
sd2 = Screwdriver()  # Another instance
sd1.operates_on  # Access class attributes
sd1.operated_by
```

```
'Screw'
'Hand'
```

In many cases, we will define a special **constructor method** named `__init__()`, which will be called at instatiation and passed any arguments provided as follows (we remove docstrings for brevity):

```python
class Screwdriver:
    operates_on = "Screw"  # Class data attributes
    operated_by = "Hand"

    def __init__(self, head, length):
        self.head = head  # Instance data attributes
        self.length = length

    def drive(self, screw, angle):  # Method definition
        return screw.turn(angle)
```

The attributes assigned to `self` in the `__init__()` method are called **instance data attributes**. The arguments `head` and `length` are required positional arguments that are assigned to the instance data attributes `head` and `length`.

Consider the following instances:

```python
sd1 = Screwdriver(head="Phillips", length=7)
sd2 = Screwdriver(head="Flat", length=8)
print(f"sd1 is a {sd1.head}head operated by {sd1.operated_by}")
print(f"sd2 is a {sd2.head}head operated by {sd2.operated_by}")
```

```
sd1 is a Phillipshead operated by Hand
sd2 is a Flathead operated by Hand
```

So we see that instances can have different instance data attributes but they share the same class data attributes.

Note that every method has as its first argument `self`, which is the conventional name given to the first argument, which is always the instance object that includes the method. When calling a method of an instance, we do not provide the `self` argument because it is provided automatically. Before we can call the `Screwdriver` method `drive()`, we should define a `Screw` class as follows:

```
class Screw:
    """Represents a screw fastener"""
    def __init__(self, head, angle=0, handed="Right"):
        self.head = head
        self.angle = angle
        self.handed = handed

    def turn(self, angle):
        """Mutates angle attribute by adding angle"""
        self.angle += angle
```

Instances of the `Screw` class have 3 instance attributes, `head`, `angle`, and `handed`. Let's instantiate a screw and give it a turn as follows:

```
s1 = Screw(head="Phillips")
print(f"Initial angle: {s1.angle}")
sd1.drive(screw=s1, angle=3)  # Turn the screw 3 units
print(f"Mutated angle: {s1.angle}")
sd1.drive(screw=s1, angle=6)  # Turn the screw 6 units
print(f"Mutated angle: {s1.angle}")
```

```
Initial angle: 0
Mutated angle: 3
Mutated angle: 9
```

As we have seen in this example, instance data attributes can represent the **state** of an object and methods can mutate or **transition** that state. This opens up a vast number of possibilities for the engineer, for we often need to keep track of the states and state transitions of objects in engineering systems.

2.5.1 Derived Classes

A **derived class** (also called a subclass) is a class that uses another class as its basis. A class that is a derived class's basis is called a **base class** for the derived class. A derived class **inherits** all of the class data attributes and methods of its base class, and it typically has additional class data attributes or methods of its own.

Continuing the screw and screwdriver example from above, let's define a derived class for representing set screws[3] as follows:

```
class SetScrew(Screw):
    """Represents a set screw fastener"""
    def __init__(self, head, tip, angle=0, handed="Right"):
        self.tip = tip  # Add instance attribute
        super().__init__(head, angle, handed)  # Call base constructor
```

3. A set screw is a screw that holds an object in place via a force applied by its tip.

The base class was specified by placing it in parentheses in `SetScrew(Screw)`. It is not necessary to define a new constructor, but to we did so to add the instance attribute `tip`. Rather than duplicating the rest of the base class's constructor, the base constructor was called via `super().__init__()`, to which its relevant arguments were passed straight through. Trying out the subclass,

```
sd3 = Screwdriver(head="Hex", length=5)
ss1 = SetScrew(head="Hex", tip="Nylon")
sd3.drive(ss1, 2)  # Drive the set screw
print(f"Set screw angle: {ss1.angle}")
```

```
Set screw angle: 2
```

Note what has occurred: the `Screwdriver` instance `sd3` used its `drive()` method to call the `turn()` method of `SetScrew` instance `ss1`. This `turn()` method was inherited from the `Screw` class, so we didn't have to repeat the definition of `turn()` in the subclass definition of `SetScrew`.

2.6 Style Conventions

As we have seen, the syntax and semantics of Python leave open many semantically equivalent choices to be made for a given program. For instance, a list can be defined with

```
l = [
    "foo",
    "bar",
    "baz"
]
```

or with

```
l = ["foo", "bar", "baz"]
```

Semantically, these are equivalent. Which is better? This is the question of **style**: what is a good way to make these decisions?

A **style guide** is a collection of rules to be applied consistently to a program, a software suite, or even all programs in a given language. Consistency is crucial; without it, a program will be harder to read, maintain, and improve. The Python standard library style guide by Rossum, Warsaw, and Coghlan (2024) (often called simply "PEP 8") has become the de facto official Python style guide. Most professionally written programs will follow this guide with more or less variation (e.g., there might be a "house" style for certain cases). However, as Emerson tells us and PEP 8 reminds us,

> A foolish consistency is the hobgoblin of little minds

A common paraphrase of this leaves off the qualifier "foolish," suggesting that any consistency should be dispatched, which would be … inconsistent … with so much wisdom that embraces the value of consistency. However, a *foolish* consistency is, indeed, a hobgoblin; a style guide is most effective when its wielder knows when and how to break from it.

Rather than presenting the PEP 8 style guide in detail, we will learn it through experience. Most of the code in this book uses the PEP 8 style, with some variation for succinct presentation. Furthermore, the reader should turn on autoformatting in the Spyder IDE by opening preferences with Ctrl + , , navigating to the tab Completion and linting, and, under the "Code formatting" section, choose the code formatting provider black. Check the box "Autoformat files on save" and click OK. Now, whenever you save a file, it will be autoformatted in conformance with the PEP 8 style guide.

The Black code formatter (Langa and contributors to Black 2024) necessarily goes beyond the PEP 8 guide, which still has some flexibility, to enforce a strict style. It is used extensively in the software development community, so beginning with this as a baseline should help you develop well in your own style.

There are some important aspects of style that are not enforced by PEP 8 or Black, including some aspects of docstrings and type hints. For these, we will follow another popular style guide from Google (2024), as described in the following sections.

2.6.1 Docstrings

A **docstring** is a string literal that is the first statement of a function definition, class definition, or module (i.e., a `.py` file). By convention, it is surrounded with three double-quotation marks, like

```
"""Here is a docstring."""
```

For simple functions, classes, and modules, this can be a single line of 88 characters or less. For instance,

```
def foo(x):
    """Return a fun string that ends with x."""
    return f"This string is fun {x}"
```

For complex functions, a multiline docstring is necessary and should be formatted as follows:

```
"""A succinct description or imperative.

A longer description or imperative.

Args:
    arg1: A description of the first argument.
    arg2: A description of the second argument. This one is going to be
        longer to show the hanging indent.

Returns:
    A description of the return value(s).

Raises:
    IOError: An error occurred doing X.
    ValueError: An error occurred doing Y.
"""
```

For complex classes, a multiline docstring should be formatted as follows:

```
"""A succinct description or imperative.

A longer description or imperative.

Attributes:
    attr1: A description of the first attribute.
    attr2: A description of the second attribute.
"""
```

For complex modules, a multiline docstring should be formatted as follows:

```
"""A succinct description or imperative.

A longer description or imperative.

Typical usage example:

    x = FooClass()
    y = x.BarFunction()
"""
```

The “typical usage example” idiom can also be added to function and class docstrings.

2.6.2 Type Hints

Unlike programming languages like C, Python is dynamically typed, meaning we can replace an object a given name refers to with an object of another type. For instance, the following is fine (but inadvisable):

```
x = 4  # An int type
x = "foo"  # A str type
```

In a statically typed language like C, a name's type is explicitly declared with a statement like `int x`. This makes it clear to the compiler, interpreter, or (human) programmer the type of objects to which it can refer.

In Python, type declarations are not required; however, **type hints** have been introduced to the language to serve a similar purpose. A type hint is an annotation of a name (variable or return value of a function) that indicates the type (i.e., class) of objects that should be stored in it. For instance, we can indicate that variable `x` should be of type `int` with the statement

```
x: int  # A type hint for variable x, stating x should be an int
x = 3  # Actually assign x
```

Similarly, a type hint can be included in an assignment statement, as in

```
x: int = 3 # An assignment of variable x with a type hint
```

The Python interpreter does not check these hints. However, a separate **type-checker** like mypy or pytype can be applied.[4] We will not use a type checker, but we will still find value in type hints as hints to programmers, ourselves most of all. Before the introduction of these hints to Python, it was common to annotate types via comments. Now we can reserve comments for more semantic descriptions.

For function definitions, it is very useful to use type checking, as follows:

```
def foo(x: int, y: complex, z: str) -> str:
    """An operation that returns the score."""
    return str(x + y) + z
```

As we can see, each argument can be annotated, as can the return value via the syntax `->`. Note that for numbers, a type annotation of `complex` indicates that the value can have type `complex`, `float`, or `int` (Rossum, Lehtosalo, and Langa 2024). Similarly, a type annotation of `float` indicates that the value can have type `float` or `int`. This is an interpretation of Python's "numeric tower" (Yasskin 2024).

4. At this point, unlike some other IDEs, Spyder doesn't have type-checking integration.

Box 2.2 Further Reading

- Rossum, Warsaw, and Coghlan (2024), the general Python style guide
- Rossum, Lehtosalo, and Langa (2024), the original type hint style guide
- Gonzalez et al. (2024), the variable annotation style guide
- Google (2024), the Google Python style guide

2.7 The Design of Programs

A program should be designed. Engineers are designers, so we should make excellent programmers. Unfortunately, many engineers fail to carefully consider the design of their programs, at least when it comes to those programs used for analysis and design. This often leads to programs that function poorly and are difficult to maintain. The costs associated with this are usually much greater than those accrued by a systematic design process.[5]

How should a program be designed? Software design methods abound; however, they are similar to the (many) design methods used for more conventional engineering products. Our familiarity with these, as engineers, may allow a cursory introduction to suffice.

A design typically begins with a **design problem**: a product (i.e., solution) is desired, one that does something (i.e., produces an output) for someone (i.e., a customer). For engineering computing programs, we engineers are often the customers, and the output is usually information for an analysis or design problem. The product or solution is the program itself. The design problem is often rather ill-defined at first, and we must question the customer about their goals throughout the design process. Often, the problem we thought we were solving at the beginning changes throughout the design process. Similarly, the program (solution) undergoes several iterations.

Two of the most important ways to think about program design are introduced in the rest of this section: (1) the functional analysis design method and (2) the pseudocode algorithm representation.

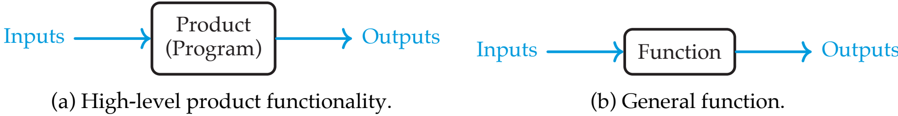
5. It should be noted, however, that for back-of-the-envolope calculations, we need not spend the time on a systematic design process. A paradigm I like to use is to create an exploratory `play.py` file in a project or simply use an interactive session. In this type of environment, we can explore without concern for structure, style, and careful design processes.

2.7.1 The Functional Analysis Design Method

One way to organize our thinking about the program (solution) is to begin at the highest level and ask the question:

> What will the program start with and what will it need to produce?

This amounts to the question: What are its **inputs** and **outputs**? Figure 2.1a illustrates this high-level conception of the program as a block that transforms inputs into outputs.



(a) High-level product functionality. (b) General function.

Figure 2.1. The functional design method (a) at the highest level and (b) in general, for any level.

This is the beginning of the **functional analysis design method**. We have treated the program as a **function**, which, like a mathematical function, maps inputs to ouputs. The next step is to consider the question:

> How can the program achieve this transformation of its inputs to its outputs?

Many techniques may be explored, but often they can be separated into **subfunctions**. The subfunctions can themselves have subfunctions. This way of breaking down the problem into functions and subfunctions is the key to the power of the functional analysis design method. We see that, at all levels, the paradigm of mapping of inputs to outputs through functions applies, as illustrated in figure 2.1b. Drawing these functional blocks and connecting their inputs and outputs is a crucial step in a program design process.

In Python program design, we have Python functions (section 1.7) and methods (section 1.3) to perform the role of the function blocks in the functional analysis design method. Inputs are passed as input arguments (or objects) and outputs are the returned values (or mutated objects). If we begin by sketching a functional diagram of inputs, outputs, and functional blocks from the highest level to the lowest, the programming of the corresponding Python functions and methods becomes a matter of implementing a structure we have already thoroughly considered. This technique is a great way to overcome the anxiety of the “blank page.”

2.7.2 Algorithm Representation via Pseudocode

At a certain depth, the functional blocks of section 2.7.1 have reached a level that is best treated as indivisible. It is not always obvious when this point is reached, but we can always iterate later. From here, simple functional blocks can be implemented directly in Python functions. For complex functions, a more complex sequence of steps may be necessary. We call a sequence of steps like this an **algorithm**.[6]

It is often useful to outline an algorithm schematically in a language we call **pseudocode**. This is a loose but programming-like language used to describe the algorithm without concern for syntax and implementation details. That is, pseudocode is used to express in structured natural language the semantics of a program without concern for its syntax in any specific programming language. The term "structured" here means some familiar programming structures—such as assignments, branches, loops, and functions—appear in pseudocode.

A **sorting algorithm** is an algorithm for sorting the elements of a list (e.g., numbers) by some metric (e.g., magnitude) such that the input list is returned ordered. There are many sorting algorithms with different efficiencies, but a relatively simple one is called **bubble sort**. For the sake of simplicity, we consider a list of n distinct numbers to be ordered such that they have increasing value. This algorithm repeatedly passes through the list, comparing adjacent elements and swapping their positions if they are out of order. After a single pass through the list, the greatest element will be in the last position because in every pairing, it is the greater. After the second pass, the second-greatest element will be in the penultimate position. After $n - 1$ passes, the list should be sorted.

In pseudocode, we can describe the algorithm more precisely, as shown in algorithm 1.

Algorithm 1 `bubble_sort_basic` pseudocode

```
function bubble_sort_basic(list)
    for i ← 0, n − 1 do                                  ▷ Repeat n times
        for j ← 0, n − i − 1 do      ▷ Pass through potentially unsorted elements
            if list[j] > list[j + 1] then
                Swap list[j] and list[j + 1]
    return list
```

Can you think of a way to improve this algorithm? Often, when we write out the algorithm in pseudocode, it becomes more clear and improvements suggest themselves.

Once the algorithm for all complex functions are written in pseudocode, it is time to implement them as Python functions or methods. The functional analysis design

6. The term "algorithm" is actually quite broad, encompassing any technique for solving a problem.

diagrams of section 2.7.1 and the pseudocode algorithms from this section will help us rationalize this process and greatly improve our programs.

Box 2.3 Further Reading

- Abelson and Sussman (2016), a classic that teaches us how to think about computer programs
- Cross (2021), an engineering design methods (not specific to software) book with formal methods and useful case examples; see especially chapter 7 on the functional design method
- Hunt and Thomas (1999), a practical approach to designing programs, filled with nuggets of wisdom

2.8 Problems

VL

Problem 2.1 WQ Write a program in a single script that meets the following requirements:

a. It imports the standard library `random` module.
b. It defines a function `rand_sub()` that defines a list of grammatical subjects (e.g., Jim, I, you, skeletons, a tiger, etc.) and returns a random subject; consider using `random.choice()` function.
c. It defines a function `rand_verb()` that defines a list of verbs in past tense (e.g., opened, smashed, ate, became, etc.) and returns a random verb.
d. It defines a function `rand_obj()` that defines a list of grammatical objects (e.g., the closet, her, crumbs, organs) and returns a random object.
e. It defines a function `rand_sen()` that returns a random subject-verb-object sentence as a string beginning with a capital letter and ending with a period.
f. It defines a function `rand_par()` that returns a random paragraph as a string composed of 3 to 5 sentences (the number of sentences should be random—consider using the `random.randint(a, b)` function that generates an `int` between `a` and `b`, inclusively). Sentences should be separated by a space `" "` character.
g. It calls `rand_par()` three times and prints the results.

Problem 2.2 SK Rewrite the program from problem 2.1 such that it meets the following requirements:

a. It defines the functions in a separate module with the file name `rand_speech_parts.py`.
b. Instead of defining the lists of subjects, verbs, and objects inside the functions, it assigns a variable to each list in the module's global namespace and accesses them from within the functions. Why is this preferable?
c. It imports the module into the main script.
d. It print three random paragraphs, as before.

Problem 2.3 YE Write a program in a single script that meets the following requirements:

a. It imports the standard library `random` module.
b. It defines a function `rand_step(x, d, ymax, wrap=True)` that returns a `float` that is the sum of `x` and a uniformly distributed random `float` between `-d` and `d`. Consider using the `random.uniform(a, b)` function that

returns a random `float` between `a` and `b`. If `wrap` is `True`, it maps a stepped value `y > ymax` to `y - ymax` and a stepped value `y < 0` to `ymax + y`. If `wrap` is `False`, it maps a stepped value `y > ymax` to `ymax` and a stepped value `y < 0` to `0`.

c. It defines a function `rand_steps(x0, d, ymax, n, wrap=True)` that returns a `list` of `n` `float`s that are sequentially stepped from `x0`. It passes `wrap` to its call to `rand_step()`.

d. It defines a function `print_slider(k, x)` that prints `k` characters, all of which are `-` except that which has index closest to `x`, for which it prints `|`. For instance, `print_slider(17, 6.8)` should print

```
-------|---------
```

Consider using the built-in `round()` function.

e. It defines a function `rand_sliders(n, k, x0=None, d=3, wrap=True)` that prints `n` random sliders of `k` characters and max step `d` starting at the index closest to `x0`, if provided, and otherwise at the index closest `k/2`.

f. It prints 25 random wrapped sliders of 44 characters with the default step range and starting point 2.

g. It prints 20 random nonwrapped sliders of 44 characters with the step range 5 and starting point 42.

Problem 2.4 8G Rewrite the program from problem 2.3 such that it meets the following requirements:

a. It defines the functions in a separate module with the file name `rand_sliding.py`.

b. It imports the module into the main script.

c. It prints 25 random wrapped sliders of 44 characters with the default step range and starting point 42.

d. It prints 20 random nonwrapped sliders of 44 characters with the step range 5 and starting point 2.

Problem 2.5 UZ Begin with the `Screwdriver`, `Screw`, and `SetScrew` class definitions of section 2.5. Add the following features:

- Improve the `Screwdriver.drive()` method to check that its head matches the screw head and raise a `TypeError` exception if they do not
- Improve the `Screw` class by adding instance attributes `pitch` that stores the thread pitch in mm and `depth` that stores the depth of the screw in its hole

- Improve the `Screw.turn()` method to mutate the `depth` based on the angle it is turned, its handing, and its thread pitch[7]
- Create a subclass `MetricScrew` from the base class `Screw` with the additional *class* data attribute `kind = "Metric"`

Test the new features of the `Screwdriver`, `Screw`, and `MetricScrew` classes with the following steps:

a. Create an instance `ms1` of `MetricScrew` with right-handedness, a flat head, initial angle 0 rad, and thread pitch 2 mm (corresponding to an M14 metric screw)
b. Create an instance `sd1` of `Screwdriver` with a flat head
c. Turn the `ms1` screw 5 complete *clockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms1`
d. Turn the `ms1` screw 3 complete *counterclockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms1`
e. Create an instance `ms2` of `MetricScrew` that is the same as `ms1`, but with *left*-handedness
f. Turn the `ms2` screw 4 complete *counterclockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2`
g. Turn the `ms2` screw 2 complete *clockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2`
h. Create an instance `sd2` of `Screwdriver` with a hex head and try to turn the `sd1` screw and catch and print the exception

Problem 2.6 VX Improve the bubble sort algorithm of algorithm 1 by adding a test that can return the list if it is sorted before completing all the loops. Implement the improved bubble sort algorithm in a program that it meets the following requirements:

a. It defines a function `bubble_sort(l: list) -> list` that implements the bubble sort algorithm.
b. It demonstrates the `bubble_sort()` function works on three different lists of numbers.
c. It demonstrates that the early return functionality, in fact, saves us from making extra passes through the list.

Problem 2.7 YS **Preprogramming work**: In this problem, *before* writing the program specified, (1) draw a functional design method diagram (see section 2.7.1) and (2) write a pseudocode for each function (see section 2.7.2).

7. A right-handed screw with thread pitch p (mm), turned clockwise an angle α (rad), advances forward $\ell = p\alpha/(2\pi)$ mm. A full turn (i.e., $\alpha = 2\pi$) advances the screw $\ell = p$ mm. Treat clockwise turns as positive angles.

Restrictions: In this problem, most of the functions you will write already exist in the standard libary module `statistics`. You may *not* use this module for this problem, but you may use others, such as the `math` module. You may also use list methods such as `sort()`. Furthermore, you may not use any external packages.

Programming: Write a program in a single script that meets the following requirements:

a. It defines a function `stats(x: list) -> dict` that computes the following basic statistics for input list `x` of real numbers:
   i. The sample mean; for a list x of n values, the sample mean m is
   $$m(x) = \frac{1}{n} \sum_{i=0}^{n-1} x_i.$$
   ii. The sample variance; the sample variance s^2 is
   $$s^2(x) = \frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - m(x))^2 .$$
   iii. The sample standard deviation; the sample standard deviation s is
   $$s(x) = \sqrt{s^2(x)}.$$
   iv. The median; the median M of a *sorted* list x of n numbers is value of the list at index $i_M = (n-1)/2$ (i.e., the middle index); more precisely,
   $$M(x) = \begin{cases} x_{i_M} & i_M \text{ is an integer} \\ \frac{1}{2}\left(x_{\lfloor i_M \rfloor} + x_{\lceil i_M \rceil}\right) & \text{otherwise} \end{cases}$$
   where $\lfloor \cdot \rfloor$ is the floor function that rounds down and $\lceil \cdot \rceil$ is the ceiling function that rounds up. So in the case that there is no middle index, the mode is the mean of the two middle values.

   The `stats()` function should return a `dict` with the keys `"mean"`, `"var"`, `"std"`, and `"median"` correspond to values for the computed sample mean, variance, standard deviation, and median.

b. It demonstrates the `stats()` function works on three different lists of numbers.

3 Numerical Analysis I: Representations, Input and Output, and Graphics

Engineering design is usually heavily supported by numerical calculations. One of the first and enduring uses of computers is to automatically perform these calculations for engineers; in fact, the first "computers" were humans who performed numerical calculations by hand, as shown in figure 3.1.

Figure 3.1. A "computer room" at the NACA (precursor to NASA) high-speed flight station in 1949 (NASA 2002).

We call engineering numerical calculations **numerical analysis**. Many programming languages and software packages have been used for numerical analysis, but by far the most popular these days are MATLAB and Python. Python's built-in data types (e.g., `list`) and functions (e.g., `sum`) can be used directly for numerical

analysis; however, for most engineering problems it is advantageous to use the ubiquitous package NumPy (Harris et al. 2020). The primary reasons this is preferred are that NumPy provides data types, functions, and methods optimized for numerical calculations, which go far beyond Python's built-in modules. In the first several sections of this chapter, we will explore NumPy's data types (most notably the array) and some of its basic functions and methods.

The numerical data represented in NumPy often originates as data from outside the program (e.g., from sensor data gathered via an experiment). Stored in files of various formats, the data must be read from computer memory[1] into the program. This is the most common kind of a program's **inputs**. On the other end, a program can have **outputs**, frequently data files written to computer memory. In this chapter, we will learn how to load input data from files and write output data to files.

Another important kind of program output is a **graphic**—usually a graph, a plot, or a chart. A graphic is often a very important result of a numerical analysis, data visualization being a key component of engineering decision making. In this chapter, we will learn how to use the Python package Matplotlib (Hunter 2007) to generate graphics from data.

3.1 Arrays

NumPy arrays are ubiquitous for representing numerical data. Like lists, arrays are mutable and can represent collections of objects. Unlike for lists, the elements of an array must all be of the same (typically numeric) type. In this section, we learn how to create and manipulate basic arrays. Throughout this book, we will assume that the NumPy package is loaded with the following statement:

```
import numpy as np
```

3.1.1 Creating Arrays

To construct a basic array (i.e., class `np.ndarray`), we often use the function `np.array()`. Although many types of objects can be passed, a list will often do, as follows:

```
x = np.array([0.29, 0.55, -0.31, -0.84, 0.97])
```

The `shape` attribute of the `np.ndarray` object is an integer tuple representing the size (i.e., length) of each of its **dimensions**. For instance, the shape of the 1-dimensional (1D) array of five elements given the name `x` above is printed with

1. The program typically reads a file stored in "secondary" (i.e., long-term) memory and loads it into "main" memory, which is faster to access for calculations. Similarly, when a program writes to a file, it stores data that is in main memory in secondary memory.

```
print(x.shape)
```

This returns `(5,)`, which indicates the array has a single dimension, called an **axis** in NumPy, with size 5.

In NumPy, 1D arrays are called **vectors**, 2D arrays are called **matrices**, and higher-dimensional arrays are called **tensors**. The mathematical objects with the same names (i.e., vectors, matrices, and tensors) are usually represented with arrays with corresponding names. A matrix can be created as follows:

```
A = np.array([
    [0, 1, 2, 3],  # First row
    [4, 5, 6, 7],  # Second row
    [8, 9, 10, 11],  # Third row
])
```

So `A.shape` is `(3, 4)` and it represents a 3×4 mathematical matrix.

We often need to create an array with a specific shape and populate it later. Perhaps the best way to do so is a function call like

```
T = np.full(shape=(5, 3), fill_value=np.nan)
```

This creates a `(5, 3)` matrix with the special `nan` (i.e., not a number) `float` as each element. Related functions `np.zeros()` and `np.ones()` can also be used to create arrays of arbitrary shape filled with `0` and `1` values, respectively. However, for an array that is to be populated subsequently, we prefer `np.full()` with `nan` elements because it is easier to notice if parts of the array have been mistakenly left unpopulated.

The `np.arange()` function is similar to the built-in `range()` function. An array of sequential numbers with integer spacing can be easily created with the `np.arange()` function. For instance,

```
np.arange(start=0, stop=10)
np.arange(0, 10)
np.arange(stop=10)
np.arange(10)
```

All these statements yield an array that prints as follows (printed arrays look like lists):

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Of course, the `start` argument is necessary for an array that starts at a value other than `0`. There is a `step` argument for `np.arange()`, and it is useful for integer steps other than the default `1`. However, for non-integer steps, we prefer a different function altogether: `np.linspace()`. For instance, the following creates a 1D array of 31 elements from 0 to 3:

```
np.linspace(start=0, stop=3, num=11)
```

This array can be printed to show the following:

```
[0., 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3.]
```

Note that by default `stop` is the last sample; `endpoint=False` would exclude it.

We make extensive use of `np.linspace()` and regular use of the similar `np.logspace()`, which creates a 1D array logarithmically spaced between two powers of the base provided. For instance, in the default base 10, we can generate 6 values from 10^0 to 10^3 with

```
np.logspace(start=0, stop=3, num=6)
```

This array can be printed to show the following:

```
[1., 3.98107171, 15.84893192, 63.09573445, 251.18864315, 1000.]
```

Most of the time, we use numeric values (i.e., `dtype`s of `int`, `float`, `complex`, and `bool` types) in Python arrays. It is also possible to create a Python **object array** with `dtype` `"O"` for object; for instance:

```
A = np.array([{"foo": "bar"}, {"bar": "baz"}])  # An object array
```

Printing the `A.dtype` attribute reveals that it is type `object`. It is occasionally advantageous to use object arrays instead of lists, primarily for the convenience of NumPy's array manipulation capabilities.

3.1.2 Accessing, Slicing, and Assigning Elements

Array elements can be accessed via indices in the same way as with lists. For instance,

```
A = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]) # 3x4
A[0, 0]  # => 0
A[0, 3]  # => 3
A[1, 0]  # => 4
A[2, 3]  # => 11
A[1]  #=> [4, 5, 6, 7]
```

Similarly, array slicing has the same syntax as list slicing. For instance,

```
A[0:2]  # => [[0, 1, 2, 3], [4, 5, 6, 7]] (view)
A[:-1]  # => [[0, 1, 2, 3], [4, 5, 6, 7]] (view)
A[:, 1]  # => [1, 5, 9] (view)
A[:, 0:2]  # => [[0, 1], [4, 5], [8, 9]] (view)
```

An important difference between list and array slicing is that, whereas in list slicing the returned list is a *copy* of a portion of the original list, in array slicing, the returned value is a **view** of a portion of the original array. An array view object, just as with `dict` view objects (see section 1.6), uses the same data as the original

object. Therefore, mutating a view object mutates its original object and vice versa. For instance, using the same `A` matrix from above,

```
a = A[:, 0]  # => [0, 4, 8] (first column view)
a[1] = 6  # Assign a new value to view element (second row)
A[0, 0] = 2  # Assign a new value to the original array
print(a)
print(A)
```

This prints

```
[2 6 8]
[[ 2  1  2  3]
 [ 6  5  6  7]
 [ 8  9 10 11]]
```

In other words, the data in `A` and `a` are the same data. To create a copy instead of a view from a slice, simply append the `copy()` method. For instance, the following array `b` is a copy of a portion of `A`, so its data are independent:

```
b = A[:, 0].copy()  # => [0, 4, 8] (first column copy)
```

It is often useful to find the indices of an array that meet some condition. Placing an array in a conditional statement returns Boolean an array of Boolean values for each element that can be used as an index for the array. For instance,

```
A = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]) # 3x4
print(A > 4)
```

prints

```
[[False, False, False, False]
 [False,  True,  True,  True]
 [ True,  True,  True,  True]]
```

This can be used as an index to select those elements that meet the condition. For instance,

```
A[A > 4]
```

returns those elements greater than `4`, as follows:

```
[ 5,  6,  7,  8,  9, 10, 11]
```

The use of the Boolean-valued array resulting from the expression `A > 4` as an index is a type of **advanced indexing** (i.e., slicing), which uses an array with data type Boolean or integer, a non-tuple sequence, or a tuple with at least one sequence object. Unlike basic slicing, which returns a view of the original array, advanced indexing always returns a copy.

Because arrays are mutable, elements can be replaced just as with lists. For instance, the statements

```
A[0, 0] = 2
A[:, 2] = 2
```

mutate `A` such that it now prints as

```
[[ 2,  1,  2,  3],
 [ 4,  5,  2,  7],
 [ 8,  9,  2, 11]]
```

Combining the conditional indexing from above with assignment, we can make assignments based on a condition. For instance, working with the same `A` array, we can coerce values above 5 to 5 as follows:

```
A[A > 5] = 5
```

Now `A` prints as

```
[[2, 1, 2, 3],
 [4, 5, 2, 5],
 [5, 5, 2, 5]]
```

3.1.3 Appending To and Concatenating Arrays

Appending an element to an array is possible with the `np.append()` function (there is no `append()` method), but its use in loops is discouraged due to the fact that it creates a new copy of the array at every call. However, in some cases it is just the right function, and it works as shown in the following code:

```
a = np.array([0, 1, 2])
np.append(a, 3)  # => [0, 1, 2, 3]
```

When needing to construct the elements of an array in a loop, it is vastly more efficient to initialize the array with `np.full()` or similar function (see section 3.1.1) before beginning the loop, using index assignment. For instance,

```
a = np.full((5,), np.nan)  # Initialize with nans
for i in range(0, len(a)):
    if i == 0:
        a[i] = 1
    else:
        a[i] = (a[i - 1] + 1) ** 2
print(f"It is {np.any(np.isnan(a))} there are nans in a:\n{a}")
```

prints

```
It is False there are nans in a:
[1.00000e+00 4.00000e+00 2.50000e+01 6.76000e+02 4.58329e+05]
```

The statement `np.any(np.isnan(a))` is a nice idiom for detecting if any `nans` remain in the array. This is a good check that we have in fact replaced all elements of the initialized array with numbers.

Array **concatenation** is the ordered collection of arrays. The `np.concatenate()` function returns a concatenation of arrays given as a tuple to its first argument. For instance,

```
a = np.array([[0, 1], [2, 3]])  # 2x2
b = np.array([[4, 5]])  # 1x2
np.concatenate((a, b))  # => [[0, 1], [2, 3], [4, 5]] (3x2)
```

The `axis` optional argument, `0` by default, determines the dimension along which the array concatenates. For instance, with the same `a` and `b` from above,

```
np.concatenate((a, b), axis=0)  # => [[0, 1], [2, 3], [4, 5]] (3x2)
np.concatenate((a, b.T), axis=1)  # => [[0, 1, 4], [2, 3, 5]] (2x3)
```

Here we have used the **transpose** array attribute, which returns a view of the array with its axes swapped (see section 3.2.1). The arrays to be concatenated must have matching dimensions except in the `axis` dimension.

Box 3.1 Further Reading

- NumPy Developers (2024c), for a basic and short introduction to NumPy

3.2 Manipulating, Operating On, and Mapping Over Arrays

In this section, we learn to manipulate, operate on, and map over NumPy arrays.

3.2.1 Array Manipulation Functions and Methods

NumPy has many powerful functions and methods for manipulating arrays. We cover only those most frequently useful to us, here; for a full list and documentation, see (NumPy Developers 2024a).

3.2.1.1 Sorting To sort an array, the `np.sort(a)` function returns a sorted copy of `a` and the `a.sort()` method will sort (mutate) `a` itself. For instance,

```
a = np.array([6, -3, 0, 9, -6])
np.sort(a)  #  => [-6, -3,  0,  6,  9] (copy)
a.sort()  # a: [-6, -3,  0,  6,  9]
```

The function and the method have the same optional arguments, the most useful of which is `axis: int`, the axis along which to sort. The default is `-1` (i.e., the last dimension).

3.2.1.2 Transposing The mathematical matrix transpose (i.e., swapping dimensions by flipping the matrix along its diagonal) can be obtained for a Python matrix via a few different techniques. The following three techniques neither mutate the original matrix nor return transposed copies; rather, they return a transposed view of the original matrix:

```
A = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]) # 3x4
A.T  # Transpose attribute (view)
A.transpose()  # Transpose method (view)
np.transpose(A)  # Transpose function (view)
```

All three transpose statements return a 4×3 array view that prints as follows:

```
[[ 0,  4,  8],
 [ 1,  5,  9],
 [ 2,  6, 10],
 [ 3,  7, 11]]
```

The original array `A` remains the same and is linked to the transposed view objects. To get a transposed copy, append the `copy()` method to any of these statements.

Unlike for matrices, a vector transpose view is no different than the original vector. However, a **row vector** (i.e., 2D array with first axis of length `1`) or a **column vector** (i.e., 2D array with second axis of length `1`) can be created by adding an axis to a vector. For instance,

```
a = np.array([0, 1, 2, 3])  # A vector
a.T  # => [0, 1, 2, 3] (same vector view)
a[np.newaxis, :]  # => [[0, 1, 2, 3]] (1x4 row vector view)
a[:, np.newaxis]  # => [[0], [1], [2], [3]] (4x1 column vector view)
```

The following are the shapes of these objects:

- `a.shape` returns `(4,)` (i.e., a 1D array of size 4)
- `a[np.newaxis, :].shape` returns `(1, 4)` (i.e., a 2D view of shape 1×4)
- `a[:, np.newaxis].shape` returns `(4, 1)` (i.e., a 2D view of shape 4×1)

Constructing row and column vectors will be important for computing mathematical matrix-vector multiplication. They can also be properly transposed back-and-forth between row and column vectors.

3.2.1.3 Reshaping Transposing, as we have seen, is one way to reshape an array. Another way is to use the `np.reshape(a: np.ndarray, newshape: tuple)` function or the equivalent method for array `a`, `a.reshape(newshape: tuple)`. Both return a view of the original array with its elements filling the newly shaped array. The argument `newshape` may be an `int`, in which case the array is flattened 1D array, or a `tuple` following the usual pattern of an array shape. The number of elements in the new view must equal that of the original array. For instance,

```
A = np.array([[0, 1, 2], [3, 4, 5]])  # A 2x3 matrix
Ar = A.reshape((3,2))  # => [[0, 1], [2, 3], [4, 5]] (3x2 view)
```

This also provides a second way of forming a row or column vector view from a 1D array; for example,

```
a = np.array([0, 1, 2])
a_row = np.reshape((1, len(a)))  # 1x3 row vector view
a_col = np.reshape((len(a), 1))  # 3x1 column vector view
```

3.2.2 Operations on Arrays and Broadcasting

The basic arithmetic operators $+$, $-$, $\times$, and $/$ can be applied to NumPy arrays with the operators `+`, `-`, `*`, and `/`, respectively. These operations are applied **element-wise** as the following example demonstrates:

```
a = np.array([0, 1, 2])
b = np.array([3, 4, 5])
a + b  # =>  [3, 4, 7]
a - b  # =>  [-3, -3, -3]
a * b  # =>  [0, 4, 10]
a / b  # =>  [0, 0.25, 0.4]
```

3.2.2.1 Broadcasting In the cases above, the array shapes matched exactly. However, it is convenient to be able to perform these types of operations on arrays of different size such that the smaller array dimensions are **broadcast** (i.e., stretched or copied) to fill in the portions of the array it is missing. The simplest case is for an operation between a 0D array (i.e., a scalar) and another array, as in the following cases:

```
a = np.array([0, 1, 2])
a + 4  # = a + [4, 4, 4] => [4, 5, 6]
a - 4  # = a - [4, 4, 4] => [-4, -3, -2]
a * 4  # = a * [4, 4, 4] => [0, 4, 8]
a / 4  # = a / [4, 4, 4] => [0, 0.25, 0.5]
```

Here the scalar `4` was broadcast to match the (larger) `a` array with shape `(3,)` and added element-wise.

Broadcasting is quite general and works for operations between arrays of many dimensions. Dimensions of two arrays are compatible if they are of equal size or if one has size 1, in which case it can be broadcast. The dimensions are compared from last to first. If one array runs out of dimensions, the rest are treated as 1. Here are some examples of compatible array dimensions in each column:

$$
\begin{array}{rrr}
3 \times 3 & 7 \times 1 \times 4 & 9 \times 7 \times 4 \\
1 \times 3 & 4 \times 4 & 7 \times 1 \\
3 \times 1 & 7 \times 4 \times 1 & 500 \times 1 \times 1 \times 4 \\
4 \times 3 \times 1 & 1 \times 4 & 9 \times 1 \times 1
\end{array}
$$

Operations on arrays with compatible dimensions will be broadcast automatically. This is not only convenient, in most cases it is also much more efficient (in terms of memory usage and computation time) than constructing the arrays or executing loops.[2] Therefore, we usually prefer broadcasting.

3.2.2.2 Matrix Multiplication Matrix multiplication can be performed with the `@` operator. For instance, consider the matrices and column vector

$$
A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}, \text{ and } \quad \boldsymbol{x} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}.
$$

Further consider the following matrix products:

$$
AB, \quad A\boldsymbol{x}, \text{ and } \quad B^\top A\boldsymbol{x}.
$$

The following code computes these products:

```
A = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])  # 3x3
B = np.array([[0, 1], [2, 3], [4, 5]])  # 3x2
x = np.array([[0], [1], [2]])  # 3x1
A @ B  # => [[10, 13], [28, 40], [46, 67]] (3x2 matrix)
A @ x  # => [[5], [14], [23]] (3x1 column vector)
B.T @ A @ x  # => [[120], [162]] (2x1 column vector)
```

The `@` operator is equivalent to the use of the `np.matmul()` function. A related function is `np.dot(a, b)`, which takes the dot product of `a` and `b`. If `a` and `b` are matrices, this is equivalent to `a @ b`. However, in this case `np.matmul()` and `a @ b` are preferred.

3.2.2.3 Other Matrix Operations We have considered matrix transposes and multiplication. Other common mathematical matrix operations include addition, subtraction, and scalar multiplication. These are element-wise operations, so we can simply use NumPy's usual `+`, `-`, and `*` operators, respectively.

The multiplicative inverse A^{-1} of a matrix A can be computed with the `np.linalg.inv()` function from the `linalg` module. For example,

```
A = np.array([[1, 0, 0], [0, 2, 0], [0, 0, 4]])  # 3x3
np.linalg.inv(A)  # => [[1., 0., 0.], [0., 0.5, 0.], [0., 0., 0.25]]
```

2. Loops are executed in broadcasting, but these are loops in the more-efficient C programming language (in which Python is written), not in Python.

If the matrix is not invertible, the exception `LinAlgError: Singular matrix` is raised.

3.2.2.4 Element-Wise Mathematical Functions NumPy has mathematical functions that automatically operate element-wise on arrays. Trigonometric functions include `np.sin()`, `np.cos()`, and `np.tan()`; exponential and logarithmic functions include `np.exp()`, `np.log()`, and `np.log10()`; hyperbolic functions include `np.sinh()`, `np.cosh()`, and `np.tanh()`; complex-number functions include `np.real()`, `np.imag()`, and `np.angle()`; rounding functions include `np.round()`, `np.ceil()`, and `np.floor()`. All these functions operate element-wise, as shown in the following example:

```
x = np.linspace(0, 2*np.pi, 5)
np.round(np.sin(x), 10)  # => [0., 1., 0., -1., -0.] (round to 10 dec.)
np.round(np.cos(x), 10)  # => [1., 0., -1., -0., 1.] (round to 10 dec.)
```

This element-wise operation is not only convenient, it is highly optimized. NumPy takes advantage of precompiled C functions for performing these operations, so they execute much faster than would a Python loop through each element. The element-wise operation is called **vectorization** (n.b., sometimes this is the term given to the sometimes-attendant optimization), and NumPy takes great advantage of this, which is one of its key features.

3.2.3 Mapping Over Arrays and Lambda Functions

As we have seen, NumPy includes many built-in functions that are vectorized (i.e., applied element-wise). Our own custom function and method definitions can (and often should) also be vectorized. Usually, nothing special is required because we can take advantage of NumPy's built-in functions and broadcasting. For instance, consider the following example, which defines a Python function corresponding to $x \mapsto \sqrt{x} - 1$:

```
def sqrt_m1(x: np.ndarray) -> np.ndaray:
    return np.sqrt(x) - 1
```

Here `np.sqrt(x)` is already vectorized and the subtraction is automatically broadcast, so our `sqrt_m1` is vectorized. Note that this will be much faster than a `for` loop through the elements of `x`.

In general, the application of a function to each element of an array (or iterable) object is called **mapping**. In plain Python, we can apply a function `f` to each element of a list `l` with the built-in function `map(f, l)`. This is effectively just a `for` loop, which is not particularly performant. Vectorization in NumPy allows us to usually avoid `for` loops or equivalent calls to `map()`.

3.2.3.1 Lambda Functions At times, it is convenient to write an **anonymous function**, often called a **lambda function**, which is a function that need not be given a name (although it can be). Mathematically, a lambda function can be expressed as, for instance,

$$x \mapsto (x+2)^3.$$

The Python syntax for a corresponding lambda function is

```
lambda x: (x + 2) ** 3
```

A lambda function can be applied directy to an argument. For instance,

```
(lambda x: (x + 2) ** 3)(1)  ## => 27
```

It can also be given a name, as in

```
f = lambda x: (x + 2) ** 3
f(1)  # => 27
```

In some ways, this defeats the purpose of the lambda function. The PEP 8 style guide discourages this use.

So when is a lambda function actually useful? One case is for applying a non-vectorized function to a list. For instance,

```
l = [1, 2, 3]
list(map(lambda x: x ** 2, l))  # => [1, 4, 9]
```

However, in this and most cases where numerical computation, it is better to use the vectorization of NumPy. In the case of a non-numerical function mapping over a list of strings, the lambda function is a good choice, as in the following case:

```
l = ["foo", "bar", "baz"]
list(map(lambda s: s.capitalize(), l))  # => ["Foo", "Bar", "Baz"]
```

3.2.3.2 Conditional Functions There are some more complex custom functions that are difficult to vectorize. An example is a function with conditions. Consider the following function:

```
def square_positive(x):
    if x > 0:
        return x ** 2
    else:
        return x
```

This function can be applied to a single number `x`, but it cannot take an array argument. One solution would be to write a `for` loop over the elements of `x`, but this would be inefficient.

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()`:

```
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

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:

```
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:

```
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:

```
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:

```
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,

```
{
    "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:

```
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:

```
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:

```
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`:

```
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

```
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:

```
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:

```
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

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.

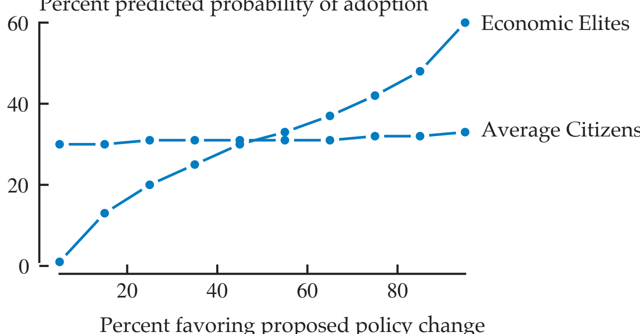


Figure 3.2. Percent predicted probability of public policy adoption for economic elites and average citizens. Study, results, and statistical model by Gilens and Page (2014).

A powerful Python package for creating graphics is Matplotlib (Hunter 2007). It is included in the base Anaconda environment and its most important module can be loaded with the following statement:

```
import matplotlib.pyplot as plt
```

In this book, we will use `plt` as the name for this module.

The rest of this section introduces the three fundamental types of graphics: function graphs, plots, and charts. Several important Matplotlib functions and methods are presented for generating each fundamental type of graphic.

3.4.1 Function Graphs

A **function graph** is a graphic that displays the relationship between a function and one or more of its arguments. A single 2D function graph can display the relationship between a function and a single argument. For instance, consider the polynomial function

$$f(x) = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0,$$

for real constant coefficients $a_0, \cdots, a_4$. To visualize the function for a given set of coefficients,

$$a_0, \cdots, a_4 = 10, 10, -20, -1, 1,$$

we could write a Python program that begins by defining the function f as a Python function as follows:

```
def f(x):
    a0, a1, a2, a3, a4 = 10, 10, -20, -1, 1
    return a4 * x ** 4 + a3 * x ** 3 + a2 * x ** 2 + a1 * x + a0
```

Our strategy is to create two NumPy arrays, one for values of x and another for corresponding values of $y = f(x)$. These values should cover the domain of interest; for instance,

```
x = np.linspace(-5, 5, 101)
y = f(x)
```

The following code will create a figure and an axis, plot `x` and `y` on the axis, set the x-axis and y-axis labels, and display the figure:

```
fig, ax = plt.subplots()  # Create a figure and an axis
ax.plot(x, y)
ax.set_xlabel("x")  # Label the x axis
ax.set_ylabel("f(x)")  # Label the y axis
plt.show()  # Display the figure
```

Consider each of the lines and what it does:

- `fig, ax = plt.subplots()` This function returns two objects with fundamental Matplotlib classes: the **figure** class `matplotlib.figure.Figure` and the **axes** class `matplotlib.axes.Axes`. Figures are the top-level containers for all elements in a Matplotlib graphic. Axes are the containers for individual plots and subplots, multiple of which can be contained in a single figure.
- `ax.plot(x, y)` This axes method plots `y` versus `x`, drawing a continuous curve connecting the points (x_i, y_i). There are many optional arguments we will later explore, but the defaults will suffice for this example.
- `ax.set_xlabel("x")` This axes method sets the x-axis label to the string argument.
- `ax.set_ylabel("f(x)")` This sets the y-axis label.
- `plt.show()` This function displays all open figures. In an IPython session (e.g., one in Spyder), this is superfluous because figures are automatically displayed in this environment.

The execution of this code displays a figure similar to the stylized version shown in figure 3.3.[4] Note that although Matplotlib connects the individual points (x_i, y_i) on the curve with straight lines, with enough points the curve appears smooth.

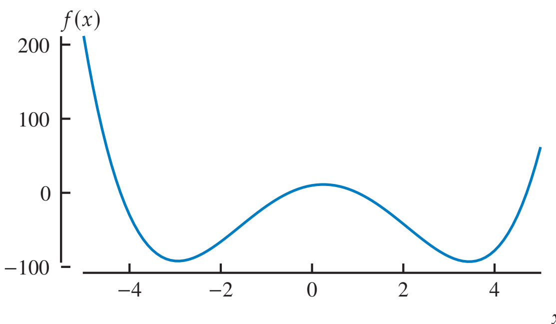


Figure 3.3. A graph of polynomial $f(x)$.

A Matplotlib figure can contain multiple axes objects and each axes object can contain multiple plots. We will explore the former in a later section and the latter in section 3.4.2.

4. We will later explore how to style and save figures. The stylization of book figures will be minimal but necessary to demonstate the aesthetic cohesion with the text for which we should strive.

3.4.2 Plots

A **plot** is a graphic that displays discrete data in relation to one or more coordinates in a coordinate system. Consider a **data set**, a set of **data points**, n-tuples $(x_{0i}, \cdots, x_{ni})$, in a **coordinate system** $(x_0, \cdots, x_n)$. A plot of the data set would display each of the data points in the data set. For instance, a plot of a data set in a Cartesian coordinate system (x, y) would display each of the data points (x_i, y_i) in the data set.

You may have observed that to create a function graph in section 3.4.1 we generated a data set of Cartesian data points (x_i, y_i) and actually created a *plot* of that data set. It turns out that a function graph is really just a plot that tries to minimize the appearance of individual data points to emphasize the continuously varying nature of the function it is presenting. On the other hand, plots that are not function graphs should usually emphasize its data points.

Experimental data are frequently presented in plots. For example, consider a data set collected in an experiment exploring the relationship among the pressure P, volume V, and temperature T of a noble gas. You may recall that noble gases are good approximations of an ideal gas, which obeys the ideal gas law

$$PV = nRT,$$

where n is the (molar) amount of gas and R is the ideal gas constant (approximately 8.314 J/(K·mol)). Our Engcom package `data` module simulates this data set; the module can be imported with the following statement:

```
import engcom.data
```

A data set can be generated for values of volume `V` and temperature `T` with the following function call:

```
d = engcom.data.ideal_gas(
    V=np.linspace(1, 2, 16),  # (m^3) Volume values
    T=np.linspace(273, 573, 4),  # (K) Temperature values
)
```

Now `d` is a dictionary with the following key–value pairs:

- `"volume"`–$V_{16\times1}$ (m^3)
- `"temperature"`–$T_{1\times4}$ (K)
- `"pressure"`–$P_{16\times4}$ (Pa)

We would like to plot P versus V for each of the 4 temperatures T; that is, plot a sequence of pairs (P_i, V_i) for each T_j. The following code loops through the temperatures and plots to the same axes object:

```
fig, ax = plt.subplots()
for j, Tj in enumerate(d["temperature"].flatten()):
    x = d["volume"]  # (m^3)
    y = d["pressure"][:,j] / 1e6  # (MPa)
    ax.plot(x, y, marker="o", color="dodgerblue")  # Circle markers
    ax.text(x=x[-1], y=y[-1], s=f"$T = {Tj}$ K")  # Label last point
```

Finally, we label the axes and display the figure with the following code:

```
ax.set_xlabel("Volume (m$^3$)")
ax.set_ylabel("Pressure (MPa)")
plt.show()
```

The figure should appear as shown in figure 3.4.

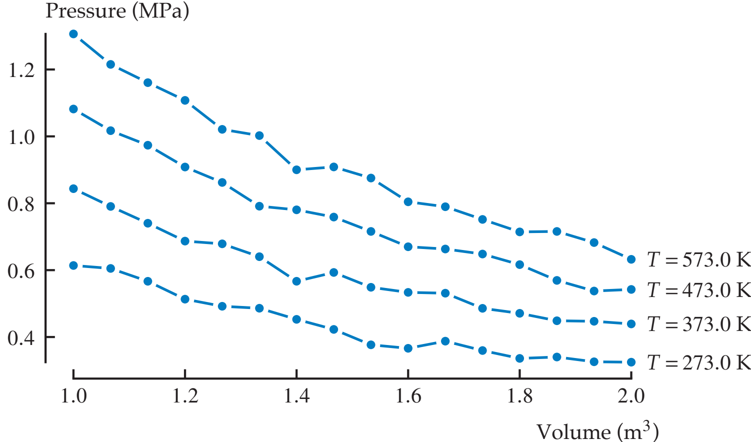


Figure 3.4. Ideal gas pressure versus volume for different temperatures.

Note the practice of labeling the individual data plots instead of creating a separate legend. At times a legend is necessary, but often it is better to simply label the data so the viewer needn't perform unnecessary work moving back-and-forth between the legend and the plot.

3.4.3 Charts

The third fundamental type of graphic is the **chart**: a data set presentation in which **signs** (i.e., icons, indices, and symbols)[5] represent data. Some signs have become commonplace for representing data:

- The **dot** • represents a data point
- The **curve** ∼ represents a continuously varying quantity or a connection between sequential data points
- The **bar** ▬ represents a quantity via its length

A function graph (section 3.4.1) represents a continuously varying quantity with a curve. A plot (section 3.4.2) represents data points with dots and connections among sequential data points with curves. The chart can use dots, curves, bars, or any other sign to represent data. Therefore, "chart" is the most general term: a function graph is a type of plot, which is a type of chart.

There are many flavors of chart in addition to the function graph and plot. Perhaps the most important are variations on the bar chart and the related histogram, covered here.

3.4.3.1 Bar Charts A **bar chart** represents and compares quantities of some type (e.g., density) for a collection of discrete **categories** (e.g., liquids). The categories may have a natural progression, in which case they should be ordered accordingly; otherwise, they should be ordered by quantity.

Consider the bar chart of thermal conductivity for various metals shown in figure 3.5. The quantity charted is thermal conductivity and the categories are types of metal. Note that not only can we easily see the conductivity of each metal, we can easily compare conductivities in this graphic. A simple table of data would be much less informative in this regard.

5. The field of **semiotics** (the study of signs) defines a sign as something that communicates a meaning. Charles Sanders Peirce distinguished three types of signs in terms of a sign's relation to its object: an **icon** has a topographical similarity with its object (e.g., ☽ is an icon representing a waxing moon), an **index** indicates something else (e.g., ↑ points to something), and a **symbol** is a sign for an object only by convention (e.g., ☣ is the biohazard symbol). The type of a given sign can be ambiguous (e.g., ☜ is an icon insofar as its object is a hand, but is an index insofar as it indicates the direction *left*).

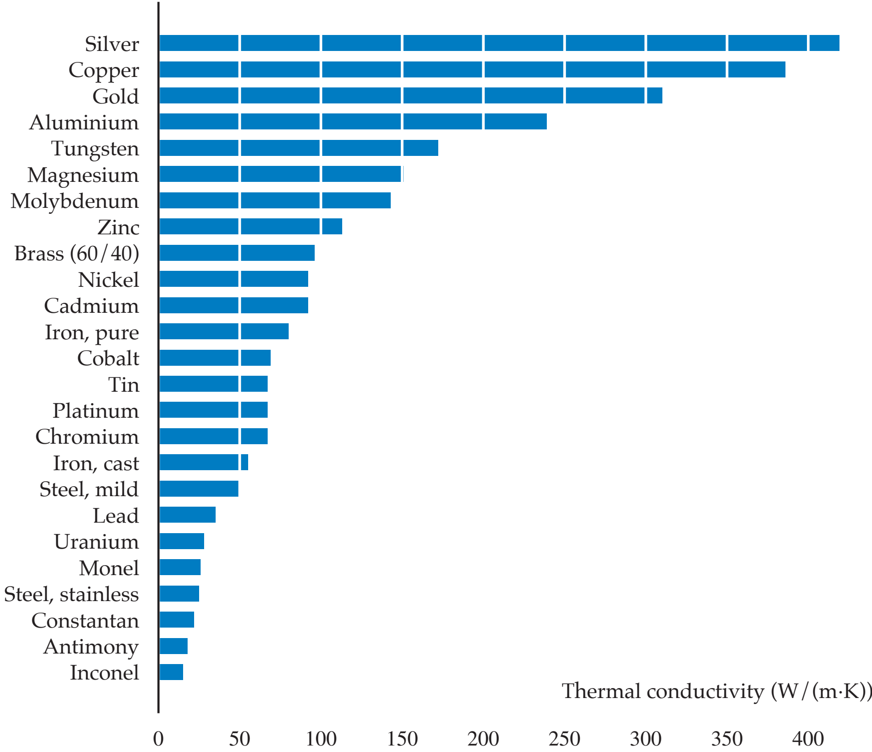


Figure 3.5. A bar chart of thermal conductivity for metals (data from Carvill (1994)).

We can produce the bar chart of figure 3.5 as follows. Begin by loading packages:

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

The data can be loaded from the `engcom.data` module as follows:

```
d = engcom.data.thermal_conductivity(category="Metals", paired=False)
y = np.arange(len(d["labels"]))
x_alpha = d["conductivity"]  # Alphabetically sorted
labels_alpha = np.array(d["labels"])  # Alphabetically sorted
```

The data is here sorted alphabetically. However, we prefer to sort it by quantity, which can be achieved with the use of the `np.lexsort()` function as follows:

```
ix = np.lexsort((labels_alpha, x_alpha))  # Sort indices
x = x_alpha[ix]
labels = labels_alpha[ix].tolist()
```

Now we can use Matplotlib's `ax.barh()` axes method (for a vertical bar chart, use `ax.bar()`) as follows:

```
fig, ax = plt.subplots()
ax.barh(y, x, color="dodgerblue")
ax.set_yticks(y, labels=labels)
ax.set_xlabel("Thermal conductivity (W/(m$\\cdot$K))")
```

In some cases we present a group of subcategories for each category, in which case a `tuple` of subcategories can be passed to `ax.barh()` or `ax.bar()`.

There are other ways to present this type of information (i.e., quantities for a collection of categories), but it is difficult to do better than the bar chart.

3.4.3.2 Histograms A **histogram** is a chart that presents a distribution of a variable. Its use of bars makes it closely related to the bar chart, but it represents the frequency a variable falls in each **bin**: an interval of values treated as a category.

Consider the histogram of my movie ratings on a 0–10 scale shown in figure 3.6. Ratings most frequently fall in the $[6, 7)$ bin. Only two movies are in the $[9, 10]$ bin. With the histogram we can easily compare the relative frequencies of values.

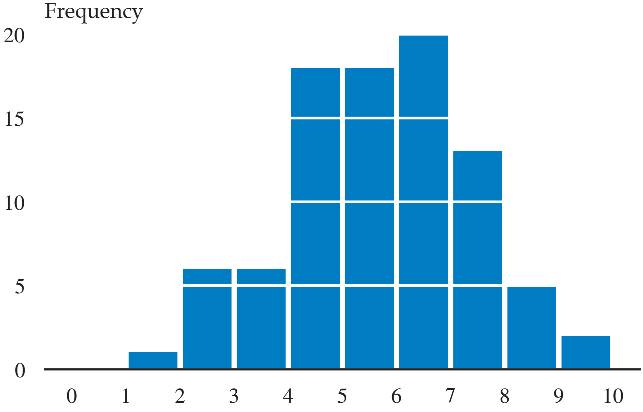


Figure 3.6. A histogram of my movie ratings on a 0–10 scale.

We can produce the histogram chart of figure 3.6 as follows. After loading the same packages as we did for the bar chart, the data can be loaded from the `engcom.data` module as follows:

```
d = engcom.data.movie_ratings_binned()
x = list(range(0,len(d["rating_freq"])))
```

Now we can use Matplotlib's `ax.bar()` axes method (for a horizontal histogram, use `ax.barh()`) as follows:

```
fig, ax = plt.subplots()
ax.bar(x, d["rating_freq"], color="dodgerblue", width=.9)
ax.set_xticks(x)
ax.set_xticklabels(d["labels"])
ax.set_xlabel("Rating out of $10$")
ax.set_ylabel("Frequency")
```

Note that Matplotlib does have a `hist()` function that can make generating histograms slightly easier. However, we prefer the flexibility of the `bar()` method.

3.4.3.3 Other Types of Charts

3.5 Problems

Problem 3.1 J3 Write a program that meets the following requirements:

a. It constructs a NumPy matrix `A` to represent the following mathematical matrix:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{bmatrix}.$$

b. It defines a function `left_up_sum(A: np.ndarray) -> np.ndarray` that adds the component (element) to the left and the component above (wrapping, if necessary) to each component. The function should pass through the array once, row-by-row, and return a new array. The function should be able to handle any size of matrix.

c. It defines a function `left_up_sums(A: np.ndarray, n: int) -> np.ndarray` that executes `left_up_sum()` `n` times and returns a new array.

d. It calls `left_up_sums()` on `A` and prints the returned array for the following values of `n`: `0`, `1`, `4`.

Problem 3.2 ZF The inner product of two real n-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined as

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i=0}^{n-1} x_i y_i.$$

The result is a scalar. The `np.inner()` and `np.dot()` functions can be used in NumPy to find the inner product of two vectors of the same size. In this problem, we will write our own function that computes the real inner product even if they are of different sizes. Write a program that meets the following requirements:

a. It defines a function

```
inner_flat_trunc(x: np.ndarray, y: np.ndarray) -> float
```

that returns the truncated inner product of vectors `a` and `b` even if the sizes of the vectors do not match by using a truncated version of the one that is too long. The function should handle any shape of input arrays by using the `flatten()` method before truncating and taking the inner product. If both input arrays do not have `dtype` attribute `np.dtype('float')` or `np.dtype('int')`, the function should raise a `TypeError` exception.

b. It calls `inner_flat_trunc()` on the following arrays:

i. A pair of arrays from the lists:
`[-1.1, 3, 2.9, -1, -9.2, 0.1]` and `[1.3, 0.2, 8.3]`

ii. An array of the integers from 0 through 13 and an array of the integers from 3 through 12
iii. An array of 21 linearly spaced elements from 0 through 10 and an array of 11 linearly spaced elements from 5 through 25.
iv. A pair of arrays of elements from the lists `[True, False, True]` and `[0, 1, 2]` (handle the exception in the main script so it runs without raising the exception)

Problem 3.3 3H Consider the following mathematical matrices and vectors:

$$A = \begin{bmatrix} 2 & 1 & 9 & 0 \\ 0 & -1 & -2 & 3 \\ -3 & 0 & 8 & -4 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 9 & -1 \\ 1 & 0 & 3 \\ 0 & -1 & 1 \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \quad \boldsymbol{y} = \begin{bmatrix} 3 & 0 & -1 \end{bmatrix}. \quad (3.1)$$

Write a program that meets the following requirements:

a. It defines NumPy arrays to represent A, B, (column vector) $\boldsymbol{x}$, and (row vector) $\boldsymbol{y}$.
b. It computes and prints the following quantities:
   i. BA
   ii. $A^\top B - 6J_{4\times 3}$, where $J_{4\times 3}$ is the 4×3 matrix of all 1 components
   iii. $B\boldsymbol{x} + \boldsymbol{y}^\top$
   iv. $\boldsymbol{x}\boldsymbol{y} + B$
   v. $\boldsymbol{y}\boldsymbol{x}$
   vi. $\boldsymbol{y}B^{-1}\boldsymbol{x}$
   vii. CB, where C is the 3×3 submatrix of the first three columns of A

Problem 3.4 DI Consider the array:

```
a = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]])  # 4x3
```

Write a program that performs and prints the results of the following operations on `a` *without* using `for` loops:

a. Adds `1` to all elements
b. Adds `1` to the last column
c. Flattens `a` to a vector
d. Reshapes `a` into a 3×4 matrix
e. Adds the vector `[1, 2, 3]` to each row
f. Adds the vector `[1, 2, 3, 4]` to each column
g. Reshapes `a` to a column vector
h. Reshapes `a` to a row vector

Problem 3.5 QX Write vectorized Python functions that operate element-wise on array arguments for the following mathematical functions:

a. $f(x) = x^2 + 3x + 9$
b. $g(x) = 1 + \sin^2 x$
c. $h(x, y) = e^{-3x} + \ln y$
d. $F(x, y) = \lfloor x/y \rfloor$
e. $G(x, y) = \begin{cases} x^2 + y^2 & \text{if } x > y \\ 2x & \text{otherwise} \end{cases}$

Problem 3.6 DN Write a program that graphs each of the following functions over the specified domain:

a. $f(x) = \tanh(4 \sin x)$ for $x \in [-5, 8]$
b. $g(x) = \sin \sqrt{x}$ for $x \in [0, 100]$
c. $h(x) = \begin{cases} 0 & \text{if } x < 0 \\ e^{-x} \sin(2\pi x) & \text{otherwise} \end{cases}$ for $x \in [-2, 6]$

Problem 3.7 WF Write a program that loads and plots ideal gas data with the `engcom.data.ideal_gas()` function in the following way:

a. The data it loads is over the volume domain: $V \in [0.1, 2.1]$ m^3
b. The data it loads has 3 temperatures: $V = 300, 400, 500$ K
c. It plots in a single graphic P versus V for each of the three temperatures
d. Each data point should be marked with a dot •
e. Sequential data points should be connected by straight lines
f. Each plot should be labeled with its corresponding temperature, either next to the plot or in a legend

Problem 3.8 Y1 Use the data from problem 3.7 to write a program that meets the following requirements:

a. It loads the pressure-volume-temperature data from problem 3.7.
b. It estimates the work W done by the gas for each of the three values of temperatures via the integral equation

$$W = -\int_{0.1}^{2.1} P(V)\, dV.$$

Note: An integral can be estimated from discrete data via the trapezoidal rule, which can be executed with NumPy's `np.trapz()` function.

c. It generates a bar chart comparing the three values of work (one for each temperature).

Problem 3.9 KG Write a program to bin data and create histogram charts that meets the following requirements:

a. It defines a function

```
binner(A: np.ndarray, nbins: int) -> (np.ndarray, np.ndarray)
```

that accepts an array `A` of data and returns an array for the frequency of the data in each bin and an array of the bin edges. Consider the following details:

i. The bin edges should include the left edge and not the right edge, except the rightmost, which should include the right edge ("left" and "right" here mean lesser and greater).
ii. The bins should be of equal width.
iii. Give a default value (e.g., `10`) for the `nbins` argument.
iv. Do *not* use the (nice) functions `np.histogram()` or `plt.hist()` for this exercise.

b. It defines a function `histogram(A: np.ndarray, nbins: int)` that calls `binner()` and `plt.bar()` to generate a histogram chart.
c. It loads all of the thermal conductivity data from the `engcom.data` module with the `engcom.data.thermal_conductivity()` function.
d. It generates 3 histograms, one for each of the following material categories (key): `"Metals"`, `"Liquids"`, and `"Gases"`. Be sure to properly label the axes.

Problem 3.10 WJ You will now create life. John Conway's Game of Life is a cellular automata game that explores the notion of life. In this problem, you will write a program for the game, which is played on a 2D grid. The grid is composed of elements called cells, each of which can be either alive or dead at a given moment. The rules of the game are simple (Johnston and Greene 2022):

- If a cell is alive, it survives to the next generation if it has 2 or 3 live neighbors; otherwise it dies.
- If a cell is dead, it comes to life in the next generation if it has exactly 3 live neighbors; otherwise it stays dead.

The neighbors of a cell are those eight cells adjacent to it (including diagonals).

Write a program for playing the game of life that meets the following requirements:

a. It defines a function

```
game_of_life(A: np.ndarray)
```

that accepts a matrix `A` that encodes the starting state for the game. Use `1` to signify an alive cell and `0` to signify a dead cell. Consider the following details:

i. The game is traditionally played on an infinite grid. However, your program should play the game of life on a torus (doughnut) made from sewing the opposite edges of the starting state `A` grid. For

instance, the neighbors above a cell in the top row are on the bottom row (i.e., neighbors wrap).

ii. A visualization is required. A very useful Matplotlib function here is `plt.matshow(A)`, which will display the numerical values of a matrix in a grid. For instance, try the following:

```
plt.matshow([[0,1,1],[1,0,1],[0,0,1]])
```

iii. Strongly consider using additional functions to define operations like "evolve one generation," "kill," "animate," and "visualize."

b. It calls `game_of_life()` on matrices corresponding to the following starting states:

i. A 5×5 grid of cells with the following pattern (blinker):

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

ii. A 20×20 grid of cells, all dead (0) except a group near the center with the following pattern (glider):

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

iii. A 40×40 grid of cells, all dead (0) except a group near the center with the pattern that can be loaded as a list from the `engcom.data` module with the function call

```
engcom.data.game_of_life_starts("gosper_glider")
```

Problem 3.11 R9 In robotic path planning, it is often important to know if a given point (e.g., a potential location of the robot) is inside of a given polygon (e.g., a shape representing an obstacle). On a plane, a polygon can be defined by a list of n points (x_i, y_i) representing the vertices of the polygon P. One algorithm for determining if a given point R is in P is called the **winding number algorithm**, which computes the winding number ω as the sum of the angles θ_i between the vectors from R to consecutive vertices P_i and P_{i+1} of the polygon, denoted $\boldsymbol{r}_i$ and $\boldsymbol{r}_{i+1}$, as shown in figure 3.7. In other words,

$$\omega = \frac{1}{2\pi} \sum_{i=0}^{n-1} \theta_i. \tag{3.2}$$

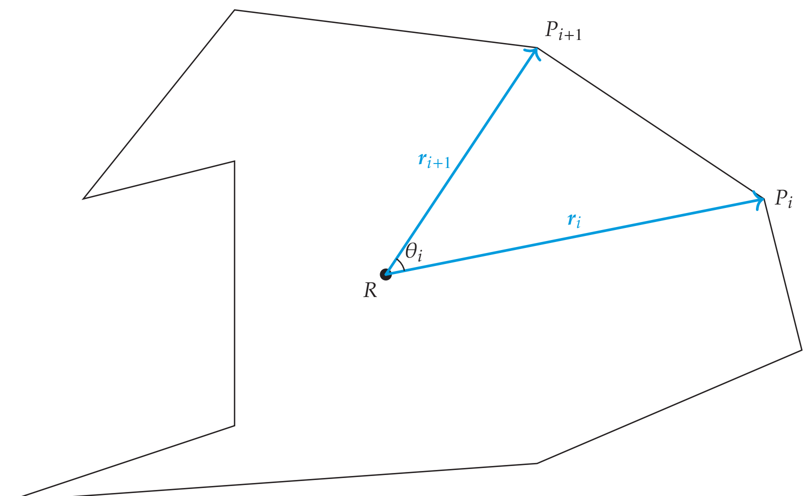


Figure 3.7. A polygon and vectors from R to two consecutive vertices.

It can be shown that if the winding number is 0, then R is outside the polygon; otherwise, it is inside. The angle ϕ_i of vector $\boldsymbol{r}_i = [r_{i_x}, r_{i_y}]$ is

$$\phi_i = \arctan(r_{i_y}/r_{i_x}),$$

where we should use `np.atan2(riy, rix)` for computation. The difference between the angles of two consecutive vectors is

$$\theta_i = \phi_{i+1} - \phi_i \text{ where } |\theta_i \leq \pi|.$$

The bound $|\theta_i \leq \pi|$ must be enforced because the acute angle is used in equation (3.2), so if $\phi_{i+1} - \phi_i < -\pi$, we should add 2π and if $\phi_{i+1} - \phi_i > \pi$, we should subtract 2π.

Write a program that meets the following requirements:

a. It defines a `Polygon` class that is constructed with instance attribute `vertices`, a list of (x_i, y_i) coordinate tuples defining the vertices of the polygon.
b. The `Polygon` class has a method `plot()` that plots the polygon as a closed curve. If a point `R` is passed to the `plot()` method, it should appear as a single point on the plot.
c. The `Polygon` class has a method `is_inside(R)` that checks if the point `R` (a tuple) is inside the polygon using a winding number algorithm. The method should return `True` if `R` is inside the polygon and `False` otherwise. Additional methods can be added to help with the computation of angles and other intermediate quantities.

d. It tests the `Polygon` class with the following polygons and points, testing if the points are inside the polygon and plotting the polygon with the points:
   i. $P = [(5,1),(2,3),(-2,3.5),(-4,1),(-2,1.5),(-2,-2),(-5,-3),(2,-2.5),(5.5,-1)]$ and the points $R_1 = (0,0)$ and $R_2 = (-4,0)$
   ii. $P = [(4,1),(1,2),(-1,1),(-4,2),(-5,-2),(-3,-2),(-5,-3),(2,-2),(5,-2)]$ and the points $R_1 = (0,0)$ and $R_2 = (-4,0)$

Restriction: Use only the NumPy and Matplotlib packages.

4 Symbolic Analysis

A **symbolic analysis**, sometimes called "analytic" as opposed to "numerical," is one that manipulates symbols called **symbolic variables**, which represent quantities. In symbolic analysis, variables of interest are solved for by means of techniques from all branches of mathematics. For engineering symbolic analysis, of particular importance are the mathematical techniques of geometry, algebra, calculus, analysis,[1] discrete mathematics, logic, set theory, probability, and statistics.

Applied to an engineering problem, the techniques of these branches of mathematics often yield **symbolic solutions** (also called "analytic" solutions), exact solutions for symbolic variables. However, there are many problems for which symbolic solutions do not exist, are unknown, are difficult to obtain, or would yield little insight into the problem (e.g., when the solution cannot be expressed simply). In such cases, the techniques of numerical analysis (chapter 5) are indicated. For those problems with nice symbolic solutions (i.e., those that can be expressed simply and can be obtained without exorbitant work), there are distinct advantages to finding symbolic solutions:

1. Symbolic solutions have provable properties (e.g., stability and bounds)
2. Symbolic solutions give designers insight into the ways design parameters affect performance (e.g., increasing the mass of this component will reduce a vibration output)
3. Symbolic solutions are much more general than numerical solutions, which are only valid for a specific set of parameters, initial conditions, boundary conditions, etc.

Computers have the ability to manipulate symbolic variables and the expressions and functions associated with them. Software designed for this purpose is called a **computer algebra system (CAS)**. Many of the techniques from the mathematics curriculum of an engineering degree are available in CASs. Popular CASs include

1. The mathematical field of analysis includes real analysis, complex analysis, differential equations, and vector analysis. Analysis developed from calculus.

Mathematica, Maple, the Symbolic Math Toolbox of MATLAB, SageMath, and the SymPy package of Python. Although most of these have an application programming interface (API) for Python, the only one that is exclusively written in and for Python is the SymPy package, and therefore we will use this as our CAS.

The SymPy package is available in the base Anaconda environment. It can be imported in a program with the following statement:

```
import sympy as sp
```

We use the alias sp throughout the text.

4.1 Symbolic Expressions, Variables, and Functions

YA

In SymPy, a **symbolic expression** is comprised of SymPy objects. Unlike numerical expressions, these are not automatically evaluated to integer or floating-point numbers. For instance, using the standard library math module, the expression math.sqrt(3)/2 immediately evaulates to the floating-point approximation of about 0.866. However, in SymPy, something else happens:[2]

```
sp.sqrt(3) / 2
```

$$\frac{\sqrt{3}}{2}$$

This is an *exact* representation of the mathematical expression, as opposed to the approximation obtained previously.

A symbolic expression can be represented as an **expression tree**:

```
sp.srepr(sp.sqrt(3) / 2)  # Show expression tree representation
```

```
'Mul(Rational(1, 2), Pow(Integer(3), Rational(1, 2)))'
```

This can be visualized as a tree graph like that shown in figure 4.1.

2. We are pretty printing results that are mathematical expressions.

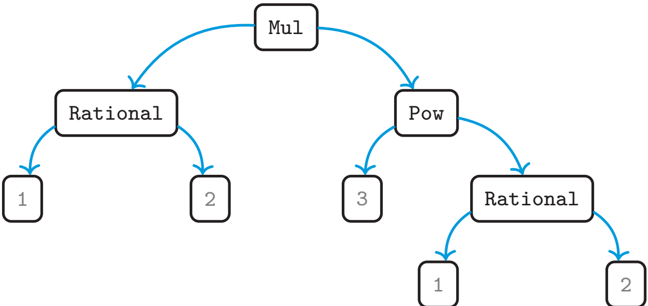


Figure 4.1. A symbolic expression tree for `sp.sqrt(3)/2`.

4.1.1 Symbolic Variables

Mathematical variables can be represented as **symbolic variables** that stand in for an unspecified number. In SymPy, symbolic variables can be created as follows:

```
x, y = sp.symbols("x, y", real=True)  # Create two real variables
```

The string passed to `sp.symbols()` can separate variables with commas and/or whitespace. The type of unspecified number being represented by the symbolic variables listed is assumed to be `complex` unless an optional argument is passed declaring otherwise. Here we have declared that `x` and `y` are real with the **predicate** `real`. Other common predicates include the following:

- Integers: `integer`, `noninteger`, `even`, and `odd`
- Real numbers: `real`, `positive`, `nonnegative`, `nonzero`, `nonpositive`, and `negative`
- Complex numbers: `complex` (default) and `imaginary`

The predicate of a symbolic variable determines the assumptions SymPy will make about it when it appears in a symbolic expression. For instance, consider the following symbolic expressions:

```
z = sp.symbols("z")  # Complex
p = sp.symbols("p", positive=True)
sp.sqrt(z**2)
sp.sqrt(x**2)  # Using real x from above
sp.sqrt(p**2)
```

↪ $\sqrt{z^2}$
↪ $|x|$
↪ p

We see that the expression automatically simplifies based on the predicates provided for each variable. This will prove especially useful once we begin using the symbolic expression manipulation techniques described in the following sections.

4.1.2 Symbolic Functions

A mathematical function can be represented in SymPy by a **symbolic function**. There are a few different ways to create these, and we will consider only the simplest and most common cases here. An **undefined function** f that should be treated as monolithic and as having no special properties can be defined as follows:

```
f = sp.Function("f")  # Type: sp.core.function.UndefinedFunction
f(x) + 3 * f(x)  # Using x from above
```

$4f(x)$

Predicates can be applied to functions, as well; for instance,

```
g = sp.Function("g", real=True)
f(x) + g(x, y) * g(3, -3)
```

$f(x)+g(3,-3)g(x,y)$

An **applied undefined function** is an undefined function that has been given an argument. For instance,

```
h = sp.Function("h")(x)  # Types: h, sp.core.function.AppliedUndef
3 * h  ## Leave off the argument
```

$3h(x)$

Undefined functions are never evaluated. At times we want to define a function that is always to be evaluated; in SymPy such a function is called a **fully evaluated function**. A fully evaluated function can be created as a regular Python function, as in the following case:

```
def F(x):
    return x**2 - 4
F(x)**2
```

$(x^2-4)^2$

For **piecewise functions**, regular Python functions with `if` statements will work, but it is preferable to use the `sp.Piecewise()` function. For instance,

```
G = sp.Piecewise(
    (x**2, x <= 0),  # x^2 for x ≤ 0
    (3*x, True)  # 3x for x > 0
)
```

Many common mathematical functions are built in to SymPy, including those shown in table 4.1.

Table 4.1: Elementary mathematical functions in SymPy.

Kind	SymPy Functions (`sp.` prefix suppressed)
Complex	`Abs(), arg(), conjugate(), im(), re(), sign()`
Trigonmetric	`sin(), cos(), tan(), sec(), csc(), cot()`
Inverse Trigonometric	`asin(), acos(), atan(), atan2(), asec(), acsc(), acot()`
Hyperbolic	`sinh(), cosh(), tanh(), coth(), sech(), csch()`
Inverse Hyperbolic	`asinh(), acosh(), atanh(), acoth(), asech()`
Integer	`ceiling(), floor(), frac() get_integer_part()`
Exponential	`exp(), log()`
Miscellaneous	`Min(), Max(), root(), sqrt()`

In rare cases, we must define a **custom function**; that is, a subclass of the `sp.Function` class. Such a function needs to have its behavior thoroughly defined. Once it is completed, it should behave just as built-in functions like `sp.sin()`. For a tutorial on writing custom functions, see SymPy Development Team (2023d).

4.2 Manipulating Symbolic Expressions

In engineering symbolic analysis, the need to manipulate, often algebraically, mathematical expressions arises constantly. SymPy has several powerful tools for manipulating symbolic expressions, the most useful of which we will consider here.

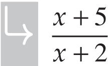
4.2.1 The `simplify()` Function and Method

A built-in Sympy function and method, `sp.simplify()`, is a common SymPy tool for manipulation because simplification is often what we want. Recall that some basic simplification occurs automatically; however, in many cases this automatic simplification is insufficient. Applying `sp.simplify()` typically results in an expression as simple as or simpler than its input; however, the precise meaning of "simpler" is quite vague, which can lead to frustrating cases in which a version of an expression we consider to be simpler is not chosen by the `sp.simplify()` algorithm. In such cases, we will often use the more manual techniques considered later in this section.

The predicates (i.e., assumptions) used to define the symbolic variables and functions that appear in a symbolic expression are respected by `sp.simplify()`. Consider the following example:

```
x = sp.symbols("x", real=True)
e0 = (x**2 + 2*x + 3*x)/(x**2 + 2*x); e0  # For display
e0.simplify()  # Returns simplified expression, leaves e0 unchanged
```

$$\frac{x^2+5x}{x^2+2x}$$



$$\frac{x+5}{x+2}$$

Note that e0 was slightly simplified automatically. The `simplify()` method further simplified by canceling an x. The use of the method does not affect the object, so it the same as the use of the function.

There are a few "knobs" to turn in the form of optional arguments to `sp.simplify()`:

- `measure` (default: `sp.count_ops()`): A function that serves as a heuristic complexity metric. The default `sp.count_ops()` counts the operations in the expression.
- `ratio` (default: `1.7`): The maximum ratio of the measures, output `out` over input `inp`, `measure(out)/measure(inp)`. Anything over `1` allows the output to be potentially more complex than the input, but it may still be simpler because the metric is just a heuristic.
- `rational` (default: `False`): By default (`False`), floating-point numbers are left alone. If `rational=True`, floating-point numbers are recast as rational numbers. If `rational=None`, floating-point numbers are recast as rational numbers during simplification, but recast to floating-point numbers in the result.
- `inverse` (default: `False`): If `True`, allows inverse functions to be cancelled in any order without knowing if the inner argument falls in the domain for which the inverse holds.[3] For instance, this allows $\arccos(\cos x) \to x$ without knowing if $x \in [0, \pi]$.
- `force` (default: `False`): If `True`, predicates (assumptions) of the variables will be ignored.

4.2.2 Polynomial and Rational Expression Manipulation

Here we consider a few SymPy functions and methods that manipulate polynomials and rational expressions.

4.2.2.1 The `expand()` Function and Method

The `expand()` function and method expresses a polynomial in the canonical form of a sum of monomials. A monomial is a polynomial with exactly one additive term. For instance,

```
sp.expand((x + 3)**2)  ## Using the real x from above
```

$x^2 + 6x + 9$

3. The usual way of defining the inverse $y = \arccos x$ is to retrict y in $x = \cos y$ to $[0, \pi]$. This is because cos is not one-to-one (e.g., $\cos 0 = \cos 2\pi = 1$), so its domain must be restricted for a proper inverse to exist. The conventional choice of domain restriction to $[0, \pi]$ is called the selection of a principal branch.

We can also expand a numerator or denominator without expanding the entire expression, as follows for $(x+3)^2/(x-2)^2$:

```
frac = (x + 3)**2/(x - 2)**2
frac.expand()
frac.expand(numer=True)
frac.expand(denom=True)
frac.expand(numer=True).expand(denom=True)
```

$$\frac{x^2}{x^2-4x+4} + \frac{6x}{x^2-4x+4} + \frac{9}{x^2-4x+4}$$

$$\frac{x^2+6x+9}{(x-2)^2}$$

$$\frac{(x+3)^2}{x^2-4x+4}$$

$$\frac{x^2+6x+9}{x^2-4x+4}$$

There are several additional options for `expand()`, including:

- `mul` (default: `True`): If `True`, distributes multiplication over addition (e.g., $5(x+1) \to 5x+5$.
- `multinomial` (default: `True`): If `True`, expands multinomial (polynomial that is not a monomial) terms into sums of monomials (e.g., $(x+y)^2 \to x^2+2xy+y^2$).
- `power_exp` (default: `True`): If `True`, expands sums in exponents to products of exponentials (e.g., $e^{3+x} \to e^3e^x$).
- `log` (default: `True`): If `True`, split log products into sums and extract log exponents to multiplicative constants (e.g., for $x, y > 0$, $\ln(x^3y) \to 3\ln x + \ln y$).
- `deep` (default: `True`): If `True`, expands all levels of the expression tree; if `False`, expands only the top level (e.g., $x(x+(y+1)^2) \to x^2 + x(y+1)^2$).
- `complex` (default: `False`): If `True`, collect real and imaginary parts (e.g., $x + y \to \Re(x) + \Re(y) + j(\Im(x) + \Im(y))$).
- `func` (default: `False`): If `True`, expand nonpolynomial functions (e.g., for the gamma function Γ, $\Gamma(x+2) \to x^2\Gamma(x) + x\Gamma(x)$).
- `trig` (default: `False`): If `True`, expand trigonometric functions (e.g., $\sin(x+y) \to \sin x \cos y - \sin y \cos x$).

4.2.2.2 The `factor()` Function and Method The `factor()` function and method returns a factorization into irreducibles factors. For polynomials, this is the reverse of `expand()`. Irreducibility of the factors is guaranteed for polynomials. Consider the following polynomial example:

```
x, y = sp.symbols("x, y", real=True)
e0 = (x + 1)**2 * (x**2 + 2*x*y + y**2); e0
e0.expand()
e0.expand().factor()
```

↳ $(x+1)^2\left(x^2+2xy+y^2\right)$
↳ $x^4+2x^3y+2x^3+x^2y^2+4x^2y+x^2+2xy^2+2xy+y^2$
↳ $(x+1)^2(x+y)^2$

Factorization can also be performed over nonpolynomial expressions, as in the following example:

```
e1 = sp.sin(x) * (sp.cos(x) + sp.sin(x))**2; e1  # Using above real x
e1.expand()
e1.expand().factor()
```

↳ $(\sin(x)+\cos(x))^2\sin(x)$
↳ $\sin^3(x)+2\sin^2(x)\cos(x)+\sin(x)\cos^2(x)$
↳ $(\sin(x)+\cos(x))^2\sin(x)$

There are two options of note:

- `deep` (default: `False`): If `True`, inner expression tree elements will also be factored (e.g., $\exp(x^2+4x+4)\rightarrow\exp((x+2)^2)$).
- `fraction` (default: `True`): If `True`, rational expressions will be combined.

An example of the latter option is given here:

```
e2 = x - 5*sp.exp(3 - x); e2  # Using real x from above
e2.factor(deep=True)
e2.factor(deep=True, fraction=False)
```

↳ $x-5e^{3-x}$
↳ $\left(xe^x-5e^3\right)e^{-x}$
↳ $x-5e^3e^{-x}$

4.2.2.3 The `collect()` Function and Method The `collect()` function and method returns an expression with specific terms collected. For instance,

```
x, y, a, b = sp.symbols("x, y, a, b", real=True)
e3 = a * x + b * x * y + a**2 * x**2 + 3 * y**2 + x * y + 8; e3
e3.collect(x)
```

↳ $a^2x^2+ax+bxy+xy+3y^2+8$
↳ $a^2x^2+x(a+by+y)+3y^2+8$

More complicated expressions can be collected as well, as in the following example:

```
e4 = a*sp.cos(4*x) + b*sp.cos(4*x) + b*sp.cos(6*x) + a * sp.sin(x); e4
e4.collect(sp.cos(4*x))
```

↳ $a\sin(x) + a\cos(4x) + b\cos(4x) + b\cos(6x)$
↳ $a\sin(x) + b\cos(6x) + (a+b)\cos(4x)$

Derivatives of an undefined symbolic function, as would appear in a differential equation, can be collected. If the function is passed to `collect()`, as in the following example, it and its derivatives are collected:

```
f = sp.Function("f")(x)  ## Applied undefined function
e5 = a*f.diff(x, 2) + a**2*f.diff(x) + b**2*f.diff(x) + a**3*f; e5
e5.collect(f)
```

↳ $a^3 f(x) + a^2 \frac{d}{dx} f(x) + a \frac{d^2}{dx^2} f(x) + b^2 \frac{d}{dx} f(x)$
↳ $a^3 f(x) + a \frac{d^2}{dx^2} f(x) + \left(a^2 + b^2\right) \frac{d}{dx} f(x)$

The `rcollect()` function (not available as a method) recursively applies `collect()`. For instance,

```
e6 = (a * x**2 + b*x*y + a*b*x)/(a*x**2 + b*x**2); e6
sp.rcollect(e6, x)  # Collects in numerator and denominator
```

↳ $\frac{abx + ax^2 + bxy}{ax^2 + bx^2}$
↳ $\frac{ax^2 + x(ab + by)}{x^2(a+b)}$

Before collection, an expression may need to be expanded via `expand()`.

4.2.2.4 The `cancel()` Function and Method The `cancel()` function and method will return an expression in the form p/q, where p and q are polynomials that have been expanded and have integer leading coefficients. This is typically used to cancel terms that can be factored from the numerator and denominator of a rational expression, as in the following example:

```
e7 = (x**3 - a**3)/(x**2 - a**2); e7
e7.cancel()
```

↳ $\frac{-a^3 + x^3}{-a^2 + x^2}$
↳ $\frac{a^2 + ax + x^2}{a + x}$

Note that there is an implicit assumption here that $x \neq a$. However, the cancelation is still valid for the limit as $x \to a$.

4.2.2.5 The `apart()` and `together()` Functions and Methods The `apart()` function and method returns a **partial fraction expansion** of a rational expression. A partial fraction expansion rewrites a ratio as a sum of a polynomial and one or

more ratios with irreducible denominators. It is of particular use for computing the inverse Laplace transform. The `together()` function is the complement of `apart()`. Here is an example of a partial fraction expansion:

```
s = sp.symbols("s")
e8 = (s**3 + 6*s**2 + 16*s + 16)/(s**3 + 4*s**2 + 10*s + 7); e8
e8.apart()  # Partial fraction expansion
e8.apart().together().cancel()  # Putting it back together
```

$$\frac{s^3 + 6s^2 + 16s + 16}{s^3 + 4s^2 + 10s + 7}$$

$$\frac{s+2}{s^2 + 3s + 7} + 1 + \frac{1}{s+1}$$

$$\frac{s^3 + 6s^2 + 16s + 16}{s^3 + 4s^2 + 10s + 7}$$

4.2.3 Trigonometric Expression Manipulation

As we saw in section 4.2.2, expressions including trigonometric terms can be manipulated with the SymPy functions and methods that are nominally for polynomial and rational expressions. In addition to these, considered here are two important SymPy functions and methods for manipulating expressions including trigonometric terms, with a focus on the trigonometric terms themselves.

4.2.3.1 The `trigsimp()` Function and Method The `trigsimp()` function and method attempts to simplify a symbolic expression via trigonometric identities. For instance, it will apply the double-angle formulas, as follows:

```
x = sp.symbols("x", real=True)
e9 = 2 * sp.sin(x) * sp.cos(x); e9
e9.trigsimp()
```

$$2\sin(x)\cos(x)$$

$$\sin(2x)$$

Here is a more involved expression:

```
e10 = sp.cos(x)**4 - 2*sp.sin(x)**2*sp.cos(x)**2 + sp.sin(x)**4; e10
e10.trigsimp()
```

$$\sin^4(x) - 2\sin^2(x)\cos^2(x) + \cos^4(x)$$

$$\frac{\cos(4x)}{2} + \frac{1}{2}$$

The hyperbolic trignometric functions are also handled by `trigsimp()`, as in the following example:

```
e11 = sp.cosh(x) * sp.tanh(x); e11
e11.trigsimp()
```

↪ $\cosh(x)\tanh(x)$
↪ $\sinh(x)$

4.2.3.2 The `expand_trig()` Function The `sp.expand_trig()` function applies the double-angle or sum identity in the expansive direction, opposite the direction of `trig_simp()`; that is,

```
e12 = sp.cos(x + y); e12
sp.expand_trig(e12)
```

↪ $\cos(x+y)$
↪ $-\sin(x)\sin(y)+\cos(x)\cos(y)$

4.2.4 Power Expression Manipulation

There are three important power identities:

$$x^a x^b = x^{a+b} \text{ for } x \neq 0, a, b \in \mathbb{C} \tag{4.1}$$

$$u^c v^c = (uv)^c \text{ for } u, v \geq 0 \text{ and } c \in \mathbb{R} \tag{4.2}$$

$$(z^d)^n = z^{dn} \text{ for } z, d \in \mathbb{C} \text{ and } n \in \mathbb{Z}. \tag{4.3}$$

Equations (4.1) to (4.3) are applied in several power expression simplification functions and methods considered here.

4.2.4.1 The `powsimp()` Function and Method The `powsimp()` function and method applies the identities of equations (4.1) and (4.2) from left-to-right (replacing the left pattern with the right). It will only apply the identity if it holds. Consider the following, applying equation (4.1):

```
x = sp.symbols("x", complex=True, nonzero=True)
a, b = sp.symbols("a, b", complex=True)
e13 = x**a * x**b; e13
e13.powsimp()
```

↪ $x^a x^b$
↪ x^{a+b}

Applying equation (4.2),

```
u, v = sp.symbols("u, v", nonnegative=True)
c = sp.symbols("c", real=True)
e14 = u**c * v**c; e14
e14.powsimp()
```

↪ $u^c v^c$
↪ $(uv)^c$

Under certain conditions (i.e., $c \in \mathbb{Q}$, a literal rational exponent), equation (4.2) is applied right-to-left automatically, so `powsimp()` appears to have no effect. For instance,

```
e15 = u**3 * v**3; e15
e15.powsimp()
```

↪ u^3v^3
↪ u^3v^3

For expressions for which the conditions for an identity does not hold, it can still be applied (at your own risk) via the `force=True` argument.

4.2.4.2 The `expand_power_exp()` and `expand_power_base()` Functions The `expand_power_exp()` function applies equation (4.1) from right-to-left (opposite of `powsimp()`), as follows:

```
e16 = x**(a + b); e16
sp.expand_power_exp(e16)
```

↪ x^{a+b}
↪ $x^a x^b$

Similarly, `expand_power_base()` applies equation (4.2) from right-to-left (opposite of `powsimp()`, as follows:

```
e17 = (u * v)**c; e17
sp.expand_power_base(e17)
```

↪ $(uv)^c$
↪ $u^c v^c$

Again, the identity will not be applied if its conditions do not hold for the expression; however, with the parameter `force=True`, it will be applied in any case.

4.2.4.3 The `powdenest()` Function The `powdenest()` function applies equation (4.3) from left-to-right. For instance,

```
z, d = sp.symbols("z, d", complex=True)
n = sp.symbols("n", integer=True)
e18 = (z**d)**n; e18
sp.powdenest(e18)
```

↪ z^{dn}
↪ z^{dn}

However, as we see from `e18`, the denesting is automatically applied. There may be situations in which `powdenest()` must still be applied manually.

4.2.5 Exponential and Logarithmic Expression Manipulation

For $x, y \geq 0$ and $n \in \mathbb{R}$, the following identities hold:

$$\log(xy) = \log(x) + \log(y) \tag{4.4}$$

$$\log(x^n) = n\log(x) \tag{4.5}$$

These can be applied with the `expand_log()` and `logcombine()` functions.

4.2.5.1 The `expand_log()` Function The `expand_log()` function applies equations (4.4) and (4.5) from left-to-right. In the following example, it applies equation (4.4):

```
x, y = sp.symbols("x, y", positive=True)
n = sp.symbols("n", real=True)
e19 = sp.log(x * y); e19
sp.expand_log(e19)
```

↪ $\log(xy)$

↪ $\log(x) + \log(y)$

In the following example, it applies equation (4.4):

```
e20 = sp.log(x**n); e20
sp.expand_log(e20)
```

↪ $\log(x^n)$

↪ $n\log(x)$

4.2.5.2 The `logcombine()` Function The `logcombine()` function applies equations (4.4) and (4.5) from right-to-left. In the following example, it applies equation (4.4):

```
e21 = sp.log(x) + sp.log(y); e21
sp.logcombine(e21)
```

↪ $\log(x) + \log(y)$

↪ $\log(xy)$

In the following example, it applies equation (4.4):

```
e22 = n * sp.log(x); e22
sp.logcombine(e22)
```

↪ $n\log(x)$

↪ $\log(x^n)$

4.2.6 Rewriting Expressions in Terms of Other Functions

At times, there are identities that can translate an expression in terms of one function (or set of functions) into an expression in terms of another function (or set of functions). In SymPy, the `rewrite()` method can perform this translation. For instance, Euler's formula, $e^{jx} = \cos x + j \sin x$ can be applied:

```
x = sp.symbols("x", complex=True)
e23 = sp.exp(1j * x); e23
e24 = e23.rewrite(sp.cos); e24  # Apply left-to-right
e24.rewrite(sp.exp)  # Apply right-to-left
```

↳ $e^{1.0ix}$
↳ $i\sin(1.0x) + \cos(1.0x)$
↳ $e^{1.0ix}$

Here is an example with a hyperbolic trigonometric function:

```
e25 = sp.tanh(x); e25
e25.rewrite(sp.exp)
```

↳ $\tanh(x)$
↳ $\dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

Finally, consider the following example with trigonometric functions:

```
x, y = sp.symbols("x, y", real=True)
e26 = sp.tan(x + y)**2; e26
e26.rewrite(sp.cos)
```

↳ $\tan^2(x + y)$
↳ $\dfrac{\cos^2\left(x + y - \frac{\pi}{2}\right)}{\cos^2(x + y)}$

4.2.7 Substituting and Replacing Expressions

One expression can be substituted for another via a few different methods, the two most useful of which are considered here.

4.2.7.1 The `subs()` Method The `subs()` method returns a copy of an expression with specific subexpressions replaced. There are three ways to specify substitutions for an expression `expr`:

- `expr.subs(old, new)`, in which `old` is replaced with `new`
- `expr.subs(iterable)`, in which `iterable` (e.g., a `list`) contains `old`/`new` pairs like `[(old0, new0), (old1, new1), ...]`
- `expr.subs(dictionary)`, in which `dictionary` contains `old`/`new` pairs like `{old0: new0, old1: new1, ...}`

Consider the following simple examples:

```
x, y, z = sp.symbols("x, y, z")
sp.sqrt(x + y).subs(x, 5)
(x + y**2 + z).subs({x: z, y: 2*z})
```

↪ $\sqrt{y+5}$

↪ $4z^2 + 2z$

By default, when an ordered iterable like a `list` or `tuple` is provided, substitutions are performed in the order given, as in the following example:

```
(x + y).subs(((x, y), (y, z)))
```

↪ $2z$

We see that the second substitution $y \to z$ is applied after the first, $x \to y$. The parameter `simultaneous`, by default `False`, can be passed as `True` so that new subexpressions are ignored by later substitutions, as in the following example:

```
(x + y).subs(((x, y), (y, z)), simultaneous=True)
```

↪ $y + z$

For dictionary substitutions, which are unordered, a canonical ordering based on the number of operations is used for reproducibility. We do not recommend relying on this canonical ordering, so if the order of substitutions is important, we recommend using an ordered iterable.

If the substitutions result in a numerical value, it will by default remain a symbolic expression:

```
sp.srepr((x + y).subs(((x, 1), (y, 3))))
```

↪ 'Integer(4)'

To get a numeric type from the result, the `evalf()` method can be used:

```
(1/y).subs(y, 3.0).evalf(n=20)  # subs() first (20 decimal places)
(1/y).evalf(subs={y: 3.0}, n=20)  # evalfr() subs (20 decimal places)
```

↪ 0.33333333333333331483

↪ 0.33333333333333333333

Note that passing the substitutions to through `evalf()` can result in a more accurate representation, so this technique is preferred. We will later [TODO: ref] return to more powerful techniques for numerical evaluation that convert SymPy expressions to numerically evaluable functions.

4.2.7.2 The `replace()` Method The `replace()` method is similar to `subs()`, but it has matching capabilities. Common usage of the `replace()` method uses **wildcard variables** of class `sp.core.symbol.Wild` that match anything in a pattern. For instance,

```
w = sp.symbols("w", cls=sp.Wild)
expr = sp.sin(x) + sp.sin(3*x)**2; expr
expr.replace(sp.sin(w), sp.cos(w)/w)
```

$$\sin(x) + \sin^2(3x)$$

$$\frac{\cos(x)}{x} + \frac{\cos^2(3x)}{9x^2}$$

Note that the wildcard variable `w` was able to match both `x` and `3*x`, and that the the wildcard could be used in the new expression as well. In this example, and in general, these replacement rules are applied without head to their validity, so they must be used with caution. For more advanced usage, see the documentation on wildcard matching, SymPy Development Team (2023b; § 6 Symbol (`sympy.core.symbol`, `Wild` class)) and the documentation for replacement, SymPy Development Team (2023b; § Basic (`sympy.core.basic.Basic`, `replace()` method)).

Box 4.1 Further Reading

- SymPy Development Team (2023c), A tutorial introduction to simplification in SymPy
- SymPy Development Team (2023a), A tutorial on advanced SymPy expression manipulation, including information about expression trees
- SymPy Development Team (2023b; § Basic (`sympy.core.basic.Basic`, `subs()` method)), SymPy documentation on the `subs()` method
- SymPy Development Team (2023b; § Basic (`sympy.core.basic.Basic`, `replace()` method)), SymPy documentation on the `replace()` method, including more advanced usage
- SymPy Development Team (2023b; 6 Symbol (`sympy.core.symbol`, `Wild` class)), SymPy documentation on the `Wild` class, including more advanced pattern matching

4.3 Solving Equations Algebraically

Virtually every engineering analysis requires the algebraic solution of an equation or a, more generally, a **system of equations** (i.e., a set of equations) to be solved simultaneously. For the engineer, this set of equations typically encodes a set of design constraints, design heuristics, and physical laws. In general, a system S of m equations in n unknown variables $x_0, \cdots, x_{n-1} \in \mathbb{C}$ and with m functions $f_0, \cdots, f_{m-1}$ can be represented as the set

$$S = \begin{cases} f_0(x_0, \cdots, x_n) = 0 \\ \vdots \\ f_m(x_0, \cdots, x_n) = 0 \end{cases}.$$

A **solution** for S is an n-tuple of values for x_i that satisfies every equation in S. There are three possible cases for a given system S of equations:

1. The system S has no solutions.
2. The system S has exactly one solution, said to be **unique**.
3. The system S has more than one solution (potentially infinitely many).

For some systems, a solution exists, but cannot be expressed in a closed-form or symbolic ("analytic") way. For such systems, a numerical solution is appropriate (see chapter 5). In some cases (e.g., n linear, independent equations and n unknown variables), a unique solution is guaranteed to exist.

There are two high-level SymPy function for solving equations algebraically, `sp.solve()` and `sp.solveset()`. The former is older, but remains the more useful for us; the latter has a simpler interface and is somewhat more mathematically rigorous, but it is often difficult to use its results programmatically. We will focus on `sp.solve()`. Neither function guarantees that it will find a solution, even if it exists, except in special cases.

Representing an equation in SymPy can be done explicitly or an expression can be treated as one side of an equation, with the other side implicitly 0. In other words, the following are equivalent ways of defining the equation $x^2 - y^2 = 2$:

```
x, y = sp.symbols("x, y")
x**2 - y**2 - 2  # == 0 Implicit equation
sp.Eq(x**2 - y**2, 2)  # Explicit equation
```

4.3.1 The sp.solve() Function

The `sp.solve()` function has the capability of solving a large class of systems of equations algebraically. The function has many optional arguments, but its basic usage is

```
sp.solve(f, *symbols, **flags)
```

Here is a basic interpretation of each argument:

- `f`: An equation or expression that is implicitly equal to zero or an iterable of equations or expressions.
- `symbols`: A symbol (e.g., variable) to solve for or an iterable of symbols.
- `flags`: Optional arguments, of which there are many. We recommend always using the `dict=True` option because it guarantees a consistent output: a list of dictionaries, one for each solution.

Consider the linear system of 3 equations and 3 unknown variables:

$$3x - 2y + 6z = -9 \tag{4.6a}$$

$$8y + 4z = -1 \tag{4.6b}$$

$$-x + 4y \quad = 0. \tag{4.6c}$$

The `sp.solve()` function can be deployed to solve this system as follows:

```
x, y, z = sp.symbols("x, y, z", complex=True)
S1 = [
  3*x - 2*y + 6*z + 9,  # == 0
  8*y + 4*z + 1, # == 0
  -x + 4*y, # == 0
]  # A system of 3 equations and 3 unknowns
sol = sp.solve(S1, [x, y, z], dict=True); sol
```

```
[{x: 15, y: 15/4, z: -31/4}]
```

Now consider a simpler system of a single equation that includes a symbolic parameter a:

$$x^2 + 3x + a.$$

Applying `sp.solve()`,

```
a = sp.symbols("a", complex=True)
S2 = [x**2 + 2*x + a]  # A system of 1 equation and 1 unknown
sol = sp.solve(S2, [x], dict=True); sol
```

```
[{x: -sqrt(1 - a) - 1}, {x: sqrt(1 - a) - 1}]
```

The quadratic formula has been applied, which yields two solutions, given in the `sol` list. Note that the solver was alerted to which symbolic variable was to be treated as an unknown variable (i.e., `x`) and which was to be treated as a known parameter (i.e., `a`) by the second argument `[x]` (i.e., `symbols`).

Suppose the solutions for x were to be substituted into an expression containing x. The `dict` object returned (here assigned to `sol`) can be used with the `subs()` method. For instance,

```
(x + 5).subs(sol[0])
(x + 5).subs(sol[1])
```

$$4-\sqrt{1-a}$$

$$\sqrt{1-a}+4$$

The `sp.solve()` function can solve for expressions and undefined functions, as well. Here we solve for an undefined function:

```
f = sp.Function("f")
eq = sp.Eq(f(x)**2 + 2*sp.diff(f(x), x), f(x))
sol = sp.solve(eq, f(x), dict=True)
```

$$f^2(x)+2\frac{d}{dx}f(x)=f(x)$$

$$f(x)=\frac{1}{2}-\frac{\sqrt{1-8\frac{d}{dx}f(x)}}{2}$$

$$f(x)=\frac{\sqrt{1-8\frac{d}{dx}f(x)}}{2}+\frac{1}{2}$$

It can solve for the derivative term, too, as follows:

```
sol = sp.solve(eq, sp.diff(f(x), x), dict=True)
```

$$\frac{d}{dx}f(x)=\frac{(1-f(x))\,f(x)}{2}$$

Example 4.1

You are designing the truss structure shown in figure 4.2. The external load of $\boldsymbol{f}_F=-f_F\hat{\boldsymbol{j}}$ (we use the standard unit vectors $\hat{\boldsymbol{i}}$, $\hat{\boldsymbol{j}}$, $\hat{\boldsymbol{k}}$), where $f_F>0$, is given. As the designer, you are to make the w dimension as long as possible under the following constraints:

- Minimize the dimension h
- The tension in all members is no more than a given T
- The compression in all members is no more than a given C
- The magnitude of the support force at pin A is no more than a given P_A
- The magnitude of the support force at pin C is no more than a given P_C

Use a static analysis and the method of joints to develop a solution for the force in each member F_{AB}, F_{AC}, etc., and the reaction forces using the sign convention that tension is positive and compression is negative. Create a function that determines

design feasibility for a given set of design parameters $\{f_F, T, C, P_A, P_C\}$ and test the function.

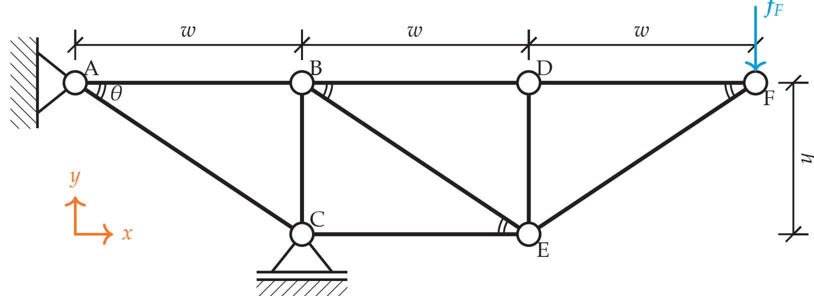


Figure 4.2. A truss with pinned joints, supported by a hinge and a floating support, with an applied force f_F.

Using the method of joints, we proceed through the joints, summing forces in the x- and y-directions. We will assume all members are in tension, and their sign will be positive if this is the case and negative, otherwise. Beginning with joint A, which includes two reaction forces R_{Ax} and R_{Ay} from the support,

$$\Sigma F_x = 0; \qquad R_{Ax} + F_{AB} + F_{AC} \cos\theta = 0 \tag{4.7}$$

$$\Sigma F_y = 0; \qquad R_{Ay} - F_{AC} \sin\theta = 0. \tag{4.8}$$

The angle θ is known in terms of the dimensions w and h as

$$\theta = \arctan\frac{h}{w}.$$

These equations can be encode symbolically as follows:

```
RAx, RAy, FAB, FAC, theta= sp.symbols(
  "RAx, RAy, FAB, FAC, theta", real=True
)
h, w = sp.symbols("h, w", positive=True)
eqAx = RAx + FAB + FAC*sp.cos(theta)
eqAy = RAy - FAC*sp.sin(theta)
theta_wh = sp.atan(h/w)
```

Proceeding to joint B,

$$\Sigma F_x = 0; \qquad -F_{AB} + F_{BD} + F_{BE} \cos\theta = 0 \tag{4.9}$$

$$\Sigma F_y = 0; \qquad -F_{BC} - F_{BE} \sin\theta = 0. \tag{4.10}$$

Encoding these equations,

```
FBD, FBE, FBC = sp.symbols("FBD, FBE, FBC", real=True)
eqBx = -FAB + FBD + FBE*sp.cos(theta)
eqBy = -FBC - FBE*sp.sin(theta)
```

For joint C, the floating support has a vertical reaction force R_C, so the analysis proceeds as follows:

$$\Sigma F_x = 0; \qquad -F_{AC}\cos\theta + F_{CE} = 0 \tag{4.11}$$

$$\Sigma F_y = 0; \qquad F_{AC}\sin\theta + F_{BC} + R_C = 0. \tag{4.12}$$

Encoding these equations,

```
FCE, RC = sp.symbols("FCE, RC", real=True)
eqCx = -FAC*sp.cos(theta) + FCE
eqCy = FAC*sp.sin(theta) + FBC + RC
```

For joint D, we can recognize that DE is a zero-force member:

$$\Sigma F_x = 0; \qquad -F_{BD} + F_{DF} = 0 \tag{4.13}$$

$$\Sigma F_y = 0; \qquad F_{DE} = 0. \tag{4.14}$$

Encoding these equations,

```
FDE, FDF = sp.symbols("FDE, FDF", real=True)
eqDx = -FBD + FDF
eqDy = FDE
```

Proceeding to joint E,

$$\Sigma F_x = 0; \qquad -F_{CE} - F_{BE}\cos\theta + F_{EF}\cos\theta = 0 \tag{4.15}$$

$$\Sigma F_y = 0; \qquad F_{BE}\sin\theta + F_{DE} + F_{EF}\sin\theta = 0. \tag{4.16}$$

Encoding these equations,

```
FEF = sp.symbols("FEF", real=True)
eqEx = -FCE - FBE*sp.cos(theta) + FEF*sp.cos(theta)
eqEy = FBE*sp.sin(theta) + FDE + FEF*sp.sin(theta)
```

Finally, consider joint F, with the externally applied force f_F,

$$\Sigma F_x = 0; \qquad -F_{DF} - F_{EF}\cos\theta = 0 \tag{4.17}$$

$$\Sigma F_y = 0; \qquad -f_F - F_{EF}\sin\theta = 0. \tag{4.18}$$

Encoding these equations,

```
fF = sp.symbols("fF", positive=True)
eqFx = -FDF - FEF*sp.cos(theta)
eqFy = -fF - FEF*sp.sin(theta)
```

In total, we have 12 force equations and 12 unknown forces (9 member forces and three reaction forces). Let's construct the system and solve it for the unkown forces, as follows:

```
S_forces = [
  eqAx, eqAy, eqBx, eqBy, eqCx, eqCy,
  eqDx, eqDy, eqEx, eqEy, eqFx, eqFy,
]  # 12 force equations
forces_unknown = [
  FAB, FAC, FBC, FBD, FBE, FCE, FDF, FDE, FEF,  # 9 member forces
  RAx, RAy, RC, # 3 reaction forces
]  # 12 unkown forces
sol_forces = sp.solve(S_forces, forces_unknown, dict=True); sol_forces
```

```
[{FAB: 2*fF*cos(theta)/sin(theta),
  FAC: -2*fF/sin(theta),
  FBC: -fF,
  FBD: fF*cos(theta)/sin(theta),
  FBE: fF/sin(theta),
  FCE: -2*fF*cos(theta)/sin(theta),
  FDE: 0,
  FDF: fF*cos(theta)/sin(theta),
  FEF: -fF/sin(theta),
  RAx: 0,
  RAy: -2*fF,
  RC: 3*fF}]
```

This solution is in terms of f_F, which is known, and θ. Because w and h are our design parameters, let's substitute `eqtheta` such that our solution is rewritten in terms of f_F, w, and h. Create a list of solutions as follows:

```
forces_wh = []  # Initialize
for force in forces_unknown:
  force_wh = force.subs(
    sol_forces[0]
  ).subs(
    theta, theta_wh
  ).simplify()
  forces_wh.append(force_wh)
  print(f"{force} = {force_wh}")
```

```
FAB = 2*fF*w/h
FAC = -2*fF*sqrt(h**2 + w**2)/h
FBC = -fF
FBD = fF*w/h
FBE = fF*sqrt(h**2 + w**2)/h
FCE = -2*fF*w/h
FDF = fF*w/h
FDE = 0
FEF = -fF*sqrt(h**2 + w**2)/h
RAx = 0
RAy = -2*fF
RC = 3*fF
```

This set of equations is excellent for design purposes. Because $f_F, w, h > 0$, the sign of each force indicates tension (+) or compression (−). For the forces with the factor w/h, clearly increasing w or decreasing h increases the force, proportionally. For the forces with the factor $\sqrt{h^2 + w^2}/h$, things are a bit more subtle. Introducing a new parameter $r = w/h$, we can rewrite these equations in a somewhat simpler manner, as follows:

```
r = sp.symbols("r", positive=True)
forces_r = []  # Initialize
force_r_subs = {}  # For substitutions
for i, force in enumerate(forces_wh):
  force_r = force.subs(w, h*r).simplify()
  forces_r.append(force_r)
  force_r_subs[forces_unknown[i]] = force_r
  print(f"{forces_unknown[i]} = {force_r}")
```

```
FAB = 2*fF*r
FAC = -2*fF*sqrt(r**2 + 1)
FBC = -fF
FBD = fF*r
FBE = fF*sqrt(r**2 + 1)
FCE = -2*fF*r
FDF = fF*r
FDE = 0
FEF = -fF*sqrt(r**2 + 1)
RAx = 0
RAy = -2*fF
RC = 3*fF
```

It is worthwhile investigating the term $\sqrt{r^2 + 1}$. Generate a graph over a reasonable range of $r = w/h$ and compare it to r and $2r$, as follows:

```
r_a = np.linspace(0, 5, 51)
fig, ax = plt.subplots()
ax.plot(r_a, np.sqrt(r_a**2 + 1), label="$\\sqrt{r^2+1}$")
ax.plot(r_a, r_a, label="$r$")
ax.plot(r_a, 2*r_a, label="$2 r$")
ax.set_xlabel("$r = w/h$")
ax.grid()
ax.legend()
```

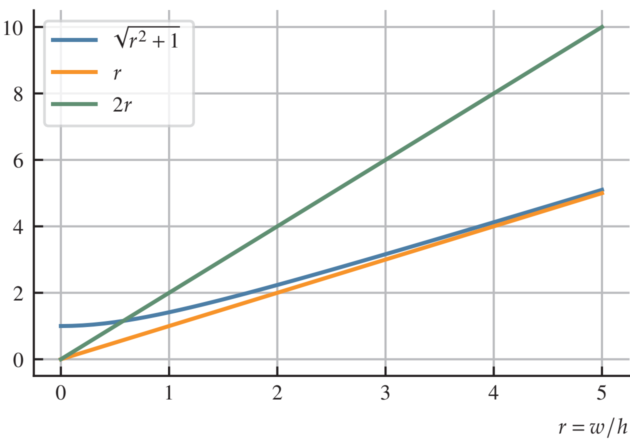


Figure 4.3. A graph of $\sqrt{r^2+1}$, where $r = w/h$.

So we see that $\sqrt{r^2+1} \to r$. That is, $r = w/h$ is the defining parameter and the design requirements are to maximize w and minimize h, which is tantamount to maximizing r. Under the reasonable assumption that $r > 1$, we can see the member with the most tension is AB, with its force $F_{AB} = 2r f_F$, and the member with the most compression is AC, with its force $F_{AC} = -2\sqrt{r^2+1} f_F$. From our design requirements, then,

$$F_{AB} = 2r f_F \leq T \tag{4.19}$$

$$-F_{AC} = 2\sqrt{r^2+1} f_F \leq C. \tag{4.20}$$

This leads to two constraints on r, call them r_T and r_C, both maxima, which can be solved for automatically as follows:

```
T, C = sp.symbols("T, C", positive=True)
eqrT = FAB.subs(force_r_subs) - T  # <= 0
eqrC = -FAC.subs(force_r_subs) - C  # <= 0
sol_rT = sp.solve(eqrT, r, dict=True)  # Solution for r_T
sol_rC = sp.solve(eqrC, r, dict=True)  # Solution for r_C
r_maxima = {
  "Tension": sol_rT[0],
  "Compression": sol_rC[0],
}
print(r_maxima)
```

```
{'Tension': {r: T/(2*fF)}, 'Compression': {r: sqrt(C**2 -
↪  4*fF**2)/(2*fF)}}
```

Another set of constraints apply to the supports. From the design requirements,

$$|\boldsymbol{R}_A| = \sqrt{R_{Ax}^2 + R_{Ay}^2} \leq P_A \tag{4.21}$$

$$|\boldsymbol{R}_C| = |R_C| \leq P_C \tag{4.22}$$

From our results above, the reaction forces don't depend on r (or w or h), so these constraints are merely to be checked to ensure that the design problem is feasible. Proceeding in a similar manner as above, we obtain two constraints on f_F, maxima f_A and f_C as follows:

```
PA, PC = sp.symbols("PA, PC", positive=True)
eqRA = sp.sqrt(RAx**2 + RAy**2).subs(force_r_subs) - PA  # <= 0
eqRC = sp.Abs(RC).subs(force_r_subs) - PC  # <= 0
sol_fFA = sp.solve(eqRA, fF, dict=True)  # Solution for f_A
sol_fFC = sp.solve(eqRC, fF, dict=True)  # Solution for f_C
load_maxima = {
  "Support A": sol_fFA[0],
  "Support C": sol_fFC[0],
}
print(load_maxima)
```

```
{'Support A': {fF: PA/2}, 'Support C': {fF: PC/3}}
```

Finally, we can create a function to perform the design, given a set of design parameters. First, define an auxilliary function to check the support constraints:

```python
def check_supports(load_maxima, design_params, report=""):
  for kLM, vLM in load_maxima.items():
    fF_design = fF.evalf(subs=design_params)
    fF_max = fF.subs(vLM).evalf(subs=design_params)
    if fF_design > fF_max:
      return False, f"Design infeasible due to {kLM} constraint: " \
        f"{fF_design:.4g} </= {fF_max:.4g}."
    else:
      report += f"{kLM} constraint satisfied: " \
        f"{fF_design:.4g} <= {fF_max:.4g}.\n"
  report += "Design feasible for supports."
  return True, report
```

Now define an auxilliary function to maximize r:

```python
def maximize_r(r_maxima, design_params, report=""):
  r_maxima_ = []  # Initialize numerical maxima
  for k_max, r_max in r_maxima.items():
    r_max_ = r.subs(r_max).evalf(subs=design_params)
    if np.abs(np.imag(complex(r_max_))) > 0.:  # Ensure real
      report += f"\nNo feasible r for {k_max} constraint."
      return False, None, report
    r_maxima_.append(r_max_)
    report += f"\nMax r for {k_max} constraint: {r_max_:.4g}."
  r_max = min(r_maxima_)  # Min of the maxima if the feasible max
  report += f"\nOverall max r = w/h = {r_max}."
  return True, r_max, report
```

Finally, define the function to design the truss:

```python
def truss_designer(load_maxima, r_maxima, design_params):
  """Returns a dict of r=w/h ratio for the truss and a report"""
  satisfied, report = check_supports(load_maxima, design_params)
  if not satisfied:
    return satisfied, report
  satisfied, r_max, report = maximize_r(
    r_maxima, design_params, report
  )
  if not satisfied:
    return satisfied, report
  return r_max, report
```

Define the three sets of design parameters in a dictionary:

```python
design_parameters_dict = {
  "1": {fF: 1000, T: 3500, C: 3200, PA: 3500, PC: 3500},
  "2": {fF: 2000, T: 4500, C: 6000, PA: 3500, PC: 3500},
  "3": {fF: 2000, T: 3500, C: 3200, PA: 6500, PC: 6000}
}  # Forces in N
```

Loop through the designs, run `truss_designer()`, and print each report:

```
for design, design_params in design_parameters_dict.items():
    r_max, rep = truss_designer(load_maxima, r_maxima, design_params)
    rep = f"Design {design} report:\n\t" + rep.replace("\n", "\n\t")
    print(rep)
```

```
Design 1 report:
  Support A constraint satisfied: 1000 <= 1750.
  Support C constraint satisfied: 1000 <= 1167.
  Design feasible for supports.
  Max r for Tension constraint: 1.750.
  Max r for Compression constraint: 1.249.
  Overall max r = w/h = 1.24899959967968.
Design 2 report:
  Design infeasible due to Support A constraint: 2000 </= 1750.
Design 3 report:
  Support A constraint satisfied: 2000 <= 3250.
  Support C constraint satisfied: 2000 <= 2000.
  Design feasible for supports.
  Max r for Tension constraint: 0.8750.
  No feasible r for Compression constraint.
```

4.4 From Symbolics to Numerics

An engineering analysis typically requires that a symbolic solution be applied via the substitution of numbers into a symbolic expression. In section 4.2.7, we considered how to subsitute numerical values into expressions using SymPy's `evalf()` method. This is fine for a single value, but frequently an expression is to be evaluated at an array of numerical values. Looping through the array and applying `evalf()` is cumbersome and computationally slow. An easier and computationally efficient technique using the `sp.lambdify()` function is introduced in this section. The function `sp.lambdify()` creates an efficient, numerically evaluable function from a SymPy expression. The basic usage of the function is as follows:

```
x = sp.symbols("x", real=True)
expr = x**2 + 7
f = sp.lambdify(x, expr)
f(2)
```

```
11
```

By default, if NumPy is present, `sp.lambdify()` vectorizes the function such that the function can be provided with NumPy array arguments and return NumPy array values. However, it is best to avoid relying on the function's implicit behavior,

which can change when different modules are present, it is best to provide the numerical module explicitly, as follows:

```
f = sp.lambdify(x, expr, modules="numpy")
f(np.array([1, 2, 3.5]))
```

```
array([ 8.  , 11.  , 19.25])
```

Multiple arguments are supported, as in the following example:

```
x, y = sp.symbols("x, y", real=True)
expr = sp.cos(x) * sp.sin(y)
f = sp.lambdify([x, y], expr, modules="numpy")
f(3, 4)
```

```
0.7492287917633428
```

All the usual NumPy broadcasting rules will apply for the function. For instance,

```
X = np.array([[1], [2]])  # 2x1 matrix
Y = np.array([[1, 2, 3]])  # 1x3 matrix
f(X, Y)
```

```
array([[ 0.45464871,  0.4912955 ,  0.07624747],
       [-0.35017549, -0.37840125, -0.05872664]])
```

Example 4.2

You are designing the circuit shown in figure 4.4. Treat the source voltage V_S, the source resistance R_S, and the overall circuit topology as known constants. The circuit design requires the selection of resistances R_1, R_2, and R_3 such that the voltage across R_3, $v_{R_3} = V_{R_3}$, and the current through R_1, $i_{R_1} = I_{R_1}$, where V_{R_3} and I_{R_1} are known constants (i.e., design requirements). Proceed through the following steps:

1. Solve for all the resistor voltages v_{R_k} and currents i_{R_k} in terms of known constants and R_1, R_2, and R_3 using circuit laws
2. Apply the constraints $v_{R_3} = V_{R_3}$ and $i_{R_1} = I_{R_1}$ to obtain two equations relating R_1, R_2, and R_3
3. Solve for R_2 and R_3 as functions of R_1 and known constants
4. Create a design graph for the selection of R_1, R_2, and R_3 given the following design parameters: $V_S = 10$ V, $R_S = 50\ \Omega$, $V_{R_3} = 1$ V, and $I_{R_1} = 20$ mA.

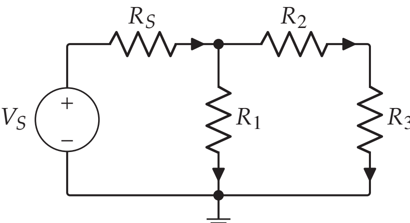


Figure 4.4. A resistor circuit design for example 4.2.

Solve for the Resistor Voltages and Currents Each resistor has an unknown voltage and current. We will develop and solve a system of equations using circuit laws. Begin by defining symbolic variables as follows:

```
v_RS, i_RS, v_R1, i_R1, v_R2, i_R2, v_R3, i_R3 = sp.symbols(
  "v_RS, i_RS, v_R1, i_R1, v_R2, i_R2, v_R3, i_R3", real=True
)
viR_vars = [v_RS, i_RS, v_R1, i_R1, v_R2, i_R2, v_R3, i_R3]
R1, R2, R3 = sp.symbols("R1, R2, R3", positive=True)
V_S, R_S, V_R3, I_R1 = sp.symbols("V_S, R_S, V_R3, I_R1", real=True)
```

There are 4 resistors, so there are $2 \cdot 4 = 8$ unknown voltages and currents; therefore, we need 8 independent equations. The first circuit law we apply is Ohm's law, which states that the ratio of voltage over current for a resistor is approximately constant. Applying this to each resistor, we obtain the following 4 equations:

```
Ohms_law = [
  v_RS - R_S*i_RS,  # == 0
  v_R1 - R1*i_R1,  # == 0
  v_R2 - R2*i_R2,  # == 0
  v_R3 - R3*i_R3,  # == 0
]
```

The second circuit law we apply is Kirchhoff's current law (KCL), which states that the sum of the current into a node must equal 0. Applying this to the upper-middle and upper-right nodes, we obtain the following 2 equations:

```
KCL = [
  i_RS - i_R1 - i_R2,  # == 0
  i_R2 - i_R3,  # == 0
]
```

The third circuit law we apply is Kirchhoff's voltage law (KVL), which states that the sum of the voltage around a closed loop must equal 0. Applying this to the left and right inner loops, we obtain the following 2 equations:

```
KVL = [
  V_S - v_R1 - v_RS,  # == 0
  v_R1 - v_R3 - v_R2, # == 0
]
```

Our collection of 8 equations are independent because none can be derived from another. They make a linear system of equations, which can be solved simultaneously as follows:

```
viR_sol = sp.solve(Ohms_law + KCL + KVL, viR_vars, dict=True)[0]
print(viR_sol)
```

$$i_{R1} = \frac{V_S (R_2 + R_3)}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$i_{R2} = \frac{R_1 V_S}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$i_{R3} = \frac{R_1 V_S}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$i_{RS} = \frac{V_S (R_1 + R_2 + R_3)}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$v_{R1} = \frac{R_1 V_S (R_2 + R_3)}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$v_{R2} = \frac{R_1 R_2 V_S}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$v_{R3} = \frac{R_1 R_3 V_S}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$v_{RS} = \frac{R_S V_S (R_1 + R_2 + R_3)}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$

Apply the Requirement Constraints The requirements that $v_{R_3} = V_{R_3}$ and $i_{R_1} = I_{R_1}$ can be encoded symbolically as two equations as follows:

```
constraints = {v_R3: V_R3, i_R1: I_R1}  # Design constraints
constraint_equations = [
  sp.Eq(v_R3.subs(constraints), v_R3.subs(viR_sol)),
  sp.Eq(i_R1.subs(constraints), i_R1.subs(viR_sol)),
]
print(constraint_equations)
```

$$V_{R3} = \frac{R_1 R_3 V_S}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$
$$I_{R1} = \frac{R_2 V_S + R_3 V_S}{R_1 R_2 + R_1 R_3 + R_1 R_S + R_2 R_S + R_3 R_S}$$

Solve for Resistances The system of 2 constraint equations and 3 unkowns (R_1, R_2, and R_3) is underdetermined, which means there are infinite solutions.

The two equations can be solved for R_1 and R_2 in terms of R_3 and parameters as follows:

```
constraints_sol = sp.solve(
  constraint_equations, [R1, R2], dict=True
)[0]
print(constraints_sol)
```

$$R_1 = -R_S + \frac{V_S}{I_{R1}} - \frac{R_S V_{R3}}{I_{R1} R_3}$$

$$R_2 = \frac{-R_3\left(I_{R1} R_S + V_{R3} - V_S\right) - R_S V_{R3}}{V_{R3}}$$

Create a Design Graph Applying the design parameters and defining numerically evaluable functions for R_1 and R_2 as functions of R_3,

```
design_params = {V_S: 10, R_S: 50, V_R3: 1, I_R1: 0.02}
R1_fun = sp.lambdify(
  [R3],
  R1.subs(constraints_sol).subs(design_params),
  modules="numpy",
)
R2_fun = sp.lambdify(
  [R3],
  R2.subs(constraints_sol).subs(design_params),
  modules="numpy",
)
```

And now we are ready to create the design graph, as follows:

```
R3_ = np.linspace(10, 100, 101)  # Values of R_3
fig, ax = plt.subplots()
ax.plot(R3_, R1_fun(R3_), label="$R_1$ ($\\Omega$)")
ax.plot(R3_, R2_fun(R3_), label="$R_2$ ($\\Omega$)")
ax.set_xlabel("$R_3$ ($\\Omega$)")
ax.legend()
ax.grid()
plt.show()
```

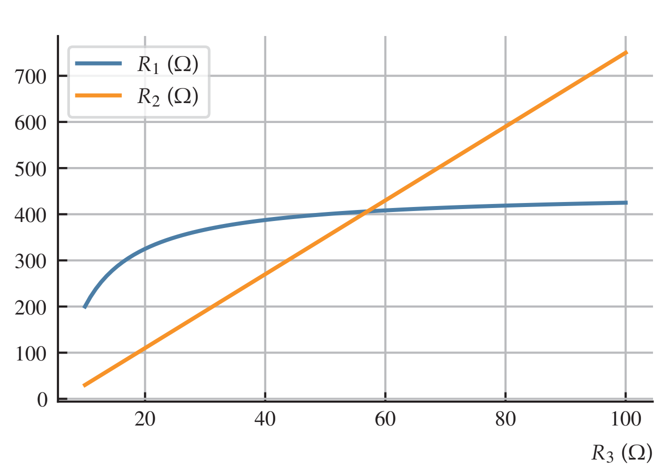


Figure 4.5. A design graph for resistors R_1, R_2, and R_3.

4.5 Vectors and Matrices

Symbolic vectors and matrices can be constructed, manipulated, and operated on with SymPy. Basic vectors and matrices are represented with the mutable `sp.matrices.dense.MutableDenseMatrix` class and can be constructed with the `sp.Matrix` constructor, as follows:

```
u = sp.Matrix([[0], [1], [2]])  # 3×1 column vector
v = sp.Matrix([[3, 4, 5]])  # 1×3 row vector
A = sp.Matrix([[0, 1, 2], [3, 4, 5], [6, 7, 8]])  # 3×3 matrix
```

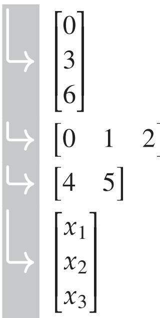
Without loss of generality, we can refer to vectors and matrices as matrices.

Symbolic variables can be elements of symbolic matrices; for instance, consider the following:

```
x1, x2, x3 = sp.symbols("x1, x2, x3")
x = sp.Matrix([[x1], [x2], [x3]])  # 3×1 vector
```

Symbolic matrix elements can be accessed with the same slicing notation as `lists` and NumPy arrays; for insance:

```
A[:,0]
A[0,:]
A[1,1:]
x[0:,0]
```



As with `list`s and contrary to arrays, these slices return a copy and not a view of the original matrix. Elements and slices can be overwritten with the same notation as `list`s and arrays, as follows:

```
A[0,0] = 7; A  # A is changed
A[:,1] = sp.Matrix([[8], [8], [8]]); A  # A is changed
```

$$\begin{bmatrix} 7 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 7 & 8 & 2 \\ 3 & 8 & 5 \\ 6 & 8 & 8 \end{bmatrix}$$

Matrix row `i` or column `j` can be deleted with the `row_del(i)` or `col_del(j)` method. These methods operate in place. For instance,

```
A.row_del(2); A
A.col_del(1); A
```

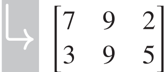
$$\begin{bmatrix} 7 & 8 & 2 \\ 3 & 8 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 7 & 2 \\ 3 & 5 \end{bmatrix}$$

Conversely, a row can be inserted at index `i` or a column can be inserted at index `j` with the method `row_insert(i, row)` or `col_insert(j, col)`. These methods do not operate in place. For instance,

```
A.row_insert(2, sp.Matrix([[9, 9]]))  # A is unchanged
A.col_insert(1, sp.Matrix([[9], [9]]))  # A is unchanged
```

$$\begin{bmatrix} 7 & 2 \\ 3 & 5 \\ 9 & 9 \end{bmatrix}$$



Addition and subtraction works element-wise, in accordance with the matrix mathematics, as follows:

```
A = sp.Matrix([[0, 1], [2, 3]])  # 2×2 matrix
B = sp.Matrix([[4, 5], [6, 7]])  # 2×2 matrix
A + B
A - B
```

$$\begin{bmatrix} 4 & 6 \\ 8 & 10 \end{bmatrix}$$

$$\begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$$

Matrix multiplication is in accordance with mathematical matrix multiplication (i.e., not element-wise), as follows:

```
A*B
B*A
```

$$\begin{bmatrix} 6 & 7 \\ 26 & 31 \end{bmatrix}$$

$$\begin{bmatrix} 10 & 19 \\ 14 & 27 \end{bmatrix}$$

The matrix inverse, if it exists, can be computed by raising the matrix to the power `-1`, as follows:

```
A**-1
B**-1
```

$$\begin{bmatrix} -\frac{3}{2} & \frac{1}{2} \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} -\frac{7}{2} & \frac{5}{2} \\ 3 & -2 \end{bmatrix}$$

The matrix transpose can be accessed as an attribute `T`, which returns a transposed copy, as follows:

```
A.T
B.T
```

$$\begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 6 \\ 5 & 7 \end{bmatrix}$$

An n-by-n identity matrix can be constructed via the `eye(n)` function, as follows:

```
sp.eye(3)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

An n-by-m matrix with all `0` compenents can be constructed via the `zeros(n, m)` function, as follows:

```
sp.zeros(2,4)
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Similarly, an n-by-m matrix with all `1` compenents can be constructed via the `ones(n, m)` function, as follows:

```
sp.ones(2,8)
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

A diagonal or block-diagonal matrix can be constructed by providing the diagonal elements to the `diag()` function, as follows:

```
D = sp.diag(1, 2, 3); D
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

The determinant of a matrix can be computed via the `det()` method, as follows:

```
D.det()
```

$$6$$

The eigenvalues and eigenvectors of a matrix can be computed via the `eigenvects()` method, which returns a list of tuples, one for each eigenvalue, of the form `(eval, m, evec)`, where `eval` is the eigenvalue, `m` is the corresponding algebraic multiplicity of the eigenvalue, and `evec` is the corresponding eigenvector. For instance,

```
A.eigenvects()
```

```
[(3/2 - sqrt(17)/2,
  1,
  [Matrix([
   [-sqrt(17)/4 - 3/4],
   [                1]])]),
 (3/2 + sqrt(17)/2,
  1,
  [Matrix([
   [-3/4 + sqrt(17)/4],
   [                1]])])]
```

4.6 Calculus

Engineering analysis regularly includes calculus. Derivatives with respect to time and differential equations (i.e., equations including derivatives) are the key mathematical models of rigid-body mechanics (e.g., statics and dynamics), solid mechanics (e.g., mechanics of materials), fluid mechanics, heat transfer, and electromagnetism. Integration is necessary for solving differential equations and computing important quantities of interest. Limits and series expansions are frequently used to in the analytic process to simplify equations and to estimate unkown quantities. In other words, calculus is central to the enterprise of engineering analysis.

4.6.1 Derivatives

In SymPy, it is possible to compute the derivative of an expression using the `diff()` function and method, as follows:

```
x, y = sp.symbols("x, y", real=True)
expr = x**2 + x*y + y**2
expr.diff(x)  # Or sp.diff(expr, x)
expr.diff(y)  # Or sp.diff(expr, y)
```

$2x + y$

$x + 2y$

Higher-order derivatives can be computed by adding the corresponding integer, as in the following second derivative:

```
expr.diff(x, 2)  # Or sp.diff(expr, x, 2)
```

2

We can see that the partial derivative is applied to a multivariate expression. The differentiation can be mixed, as well, as in the following example:

```
expr = x * y**2/(x**2 + y**2)
expr.diff(x, 1, y, 2).simplify()  # ∂^3/∂x∂y^2
```

$$\frac{2x^2\left(-x^4+14x^2y^2-9y^4\right)}{x^8+4x^6y^2+6x^4y^4+4x^2y^6+y^8}$$

The option `evaluate=False` will leave the derivative unevaluated until the `doit()` method is called, as in the following example:

```
expr = sp.sin(x)
expr2 = expr.diff(x, evaluate=False); expr2
expr2.doit()
```

$$\frac{d}{dx}\sin(x)$$

$$\cos(x)$$

The derivative of an undefined function is left unevaluated, as in the following case:

```
f = sp.Function("f", real=True)
expr = 3*f(x) + f(x)**2
expr.diff(x)
```

$$2f(x)\frac{d}{dx}f(x)+3\frac{d}{dx}f(x)$$

As we can see, the chain rule of differentiation was applied automatically.

Differentiation works element-wise on matrices and vectors, just as it works mathematically. For instance,

```
v = sp.Matrix([[x**2], [x*y]])
v.diff(x)
```

$$\begin{bmatrix}2x\\ y\end{bmatrix}$$

4.6.2 Integrals

To a symbolic integral in SymPy, use the `integrate()` function or method. For an indefinite integral, pass only the variable over which to integrate, as in

```
x, y = sp.symbols("x, y", real=True)
expr = x + y
expr.integrate(x)  # Or sp.integrate(expr, x); ∫ x + y dx
```

$$\frac{x^2}{2}+xy$$

Note that no constant of integration is added, so you may need to add your own.

The definite integral can be computed by providing a triple, as in the following example,

```
sp.integrate(expr, (x, 0, 3))  # ∫_0^3 x + y dx
sp.integrate(expr, (x, 1, y))  # ∫_1^y x + y dx
```

$$3y + \frac{9}{2}$$

$$\frac{3y^2}{2} - y - \frac{1}{2}$$

Multiple integrals can be computed in a similar fashion, as in the following examples:

```
sp.integrate(expr, (x, 0, 4), (y, 2, 3))  # ∫_2^3 ∫_0^4 x + y dxdy
```

$$18$$

To create an unevaluated integral object, use the `sp.Integral()` constructor. To evaluate an unevaluated integral, use the `doit()` method, as follows:

```
expr2 = sp.Integral(expr, x); expr2  # Unevaluated
expr2.doit()  # Evaluate
```

$$\int (x + y)\, dx$$

$$\frac{x^2}{2} + xy$$

Integration works over piecewise functions, as in the following example:

```
f = sp.Piecewise((0, x < 0), (1, x >= 0)); f
sp.integrate(f, (x, -5, 5))
```

$$\begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

$$5$$

The `integrate()` function and method is very powerful, but it may not be able to integrate some functions. In such cases, it returns an unevaluated integral.

4.6.3 Limits

In SymPy, a limit can be computed via the `limit()` function and method. The $\lim_{x \to 0}$ can be computed as follows:

```
sp.limit(sp.tanh(x)/x, x, 0)  # lim_{x→0} tanh(x)/x
```

$$1$$

The limit to infinity or negative infinity can be denoted using the `sp.oo` symbol, as follows:

```
sp.limit(2  - x * sp.exp(-x), x, sp.oo)  # lim_{x→∞}(1 − xe^{−x})
```

$$2$$

The limit can be left unevaluated using the `sp.Limit()` constructor, as follows:

```
lim = sp.Limit(2  - x * sp.exp(-x), x, sp.oo); expr  # Unevaluated
lim.doit()  # Evaluate
```

↳ $x + y$
↳ 2

The limit can be taken from a direction using the optional fourth argument, as follows:

```
expr = 1/x
lim_neg = sp.Limit(expr, x, 0, "-"); lim_neg
lim_pos = sp.Limit(expr, x, 0, "+"); lim_pos
lim_neg.doit()
lim_pos.doit()
```

↳ $\lim_{x \to 0^-} \frac{1}{x}$
↳ $\lim_{x \to 0^+} \frac{1}{x}$
↳ $-\infty$
↳ ∞

4.6.4 Taylor Series

A Taylor series (i.e., Taylor expansion) is an infinite power series approximation of an infinitely differentiable function near some point. For a function $f(x)$, the Taylor series at point x_0 is given by

$$\sum_{n=0}^{\infty} \frac{f^{(n)}}{n!}(x - x_0)^n = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots .$$

We often represent terms with power order m and greater with the **big-O notation** $O((x - x_0)^m)$. For instance, for an expansion about $x_0 = 0$,

$$\sum_{n=0}^{\infty} \frac{f^{(n)}}{n!}(x)^n = f(0) + f'(x_0)(x - x_0) + O(x^2).$$

In SymPy, the Taylor series can be found via the `series()` function or method. For instance,

```
f = sp.sin(x)
f.series(x0=0, n=4)  # Or sp.series(f, x0=0, n=4)
```

↳ $x - \frac{x^3}{6} + O\left(x^4\right)$

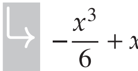
The `sp.O()` function, which appears in this result, automatically absorbs higher-order terms. For instance,

```
x**2 + x**4 + x** 5 + sp.O(x**4)
```

↳ $x^2 + O\left(x^4\right)$

To remove the `sp.O()` function from an expression, call the `removeO()` method, as follows:

```
f.series(x0=0, n=4).removeO()
```



$$-\frac{x^3}{6} + x$$

Removing the higher-order terms is frequently useful when we would like to use the nth-order **Taylor polynomial**, a truncated Taylor series, as an approximation of a function.

4.7 Solving Ordinary Differential Equations

Engineering analysis regularly includes the solution of differential equations. **Differential equations** are those equations that contain derivatives. An **ordinary differential equation (ODE)** is a differential equation that contains only ordinary, as opposed to partial, derivatives. A **linear ODE**—one for which constant multiples and sums of solutions are also solutions—is an important type that represent **linear, time-varying (LTV) systems**. For this class of ODEs, it has been proven that for a set of initial conditions, a unique solution exists (Kreyszig 2010; p. 108).

A **constant-coefficient, linear ODE** can represent **linear, time-invariant (LTI) systems**. An LTV or LTI system model can be represented as a scalar nth-order ODE, or as a system of n 1st-order ODEs. As a scalar nth-order linear ODE, with independent time variable t, output function $y(t)$, forcing function $f(t)$, and constant coefficients a_i, has the form

$$y^{(n)}(t) + a_{n-1}y^{(n-1)}(t) + \cdots a_1 y'(t) + a_0 y(t) = f(t). \tag{4.23}$$

The forcing function $f(t)$ can be written as a linear combination of derivatives of the input function $u(t)$ with $m + 1 \leq n + 1$ constant coefficients b_j, as follows:

$$f(t) = b_m u^{(m)}(t) + b_{m-1}u^{(m-1)}(t) + \cdots + b_1 u'(t) + b_0 u(t).$$

Alternatively, the same LTI system model can be represented by a system of n 1st-order ODEs, which can be written in vector form as

$$\boldsymbol{x}'(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t) \tag{4.24a}$$

$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}\boldsymbol{u}(t), \tag{4.24b}$$

where $\boldsymbol{x}(t)$ is called the state vector, $\boldsymbol{u}(t)$ is called the input vector, and $\boldsymbol{y}(t)$ is called the output vector (they are actually vector-valued functions of time), and $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ are matrices containing constants derived from system parameters (e.g., a mass, a spring constant, a capacitance, etc.). Equation (4.24) is called an LTI **state-space model**, and it is used to model a great many engineering systems.

Solving ODEs and systems of ODEs is a major topic of mathematical engineering analysis. It is typically the primary topic of one required course and a secondary topic of several others. Understanding when these solutions exist, whether they are unique, and how they can be found adds much to the understanding of engineering systems. However, it is also true that CASs such as SymPy offer the engineer excellent tools for making quick and adaptable work of this task.

Consider the ODE

$$3y'(t) + y(t) = f(t),$$

where the forcing function $f(t)$ is defined piecewise as

$$f(t) = \begin{cases} 0 & t < 0 \\ 1 & t \geq 0. \end{cases}$$

The SymPy `dsolve()` function can find the **general solution** (i.e., a family of solutions for any initial conditions) with the following code:

```
t = sp.symbols("t", nonnegative=True)
y = sp.Function("y", real=True)
f = 1  # Or sp.Piecewise(), but t≥0 already restricts f(t)
ode = sp.Eq(3*y(t).diff(t) + y(t), f)  # Define the ODE
sol = sp.dsolve(ode, y(t)); sol  # Solve
```

$$y(t) = C_1 e^{-\frac{t}{3}} + 1$$

The solution is returned as an `sp.Eq()` equation object. Note the unknown constant C_1 in the solution. To find the **specific solution** (i.e., the general solution with the initial condition applied to determine C_1) for a given initial condition $y(0) = 5$,

```
sol = sp.dsolve(ode, y(t), ics={y(0): 5}); sol
```

$$y(t) = 1 + 4e^{-\frac{t}{3}}$$

Now consider the ODE

$$y''(t) + 5y'(t) + 9y(t) = 0.$$

The SymPy `dsolve()` function can find the general solution with the following code:

```
ode = sp.Eq(y(t).diff(t, 2) + 5*y(t).diff(t) + 9*y(t), 0)
sol = sp.dsolve(ode, y(t)); sol
```

$$y(t) = \left(C_1 \sin\left(\frac{\sqrt{11}t}{2}\right) + C_2 \cos\left(\frac{\sqrt{11}t}{2}\right)\right) e^{-\frac{5t}{2}}$$

This is a decaying sinusoid. Applying two initial conditions, $y(0) = 4$ and $y'(0) = 0$, we obtain the following:

```
sol = sp.dsolve(
    ode, y(t),
    ics={y(0): 4, y(t).diff(t).subs(t, 0): 0}
); sol
```

$$y(t) = \left(\frac{20\sqrt{11}\sin\left(\frac{\sqrt{11}t}{2}\right)}{11} + 4\cos\left(\frac{\sqrt{11}t}{2}\right)\right)e^{-\frac{5t}{2}}$$

We see here that to apply the initial condition $y'(0) = 0$, the derivative must be applied before substituting $t \to 0$.

Solving sets (i.e., systems) of first-order differential equations is similar. Consider the set of differential equations

$$y_1'(t) = y_2(t) - y_1(t) \text{ and } y_2'(t) = y_1(t) - y_2(t).$$

To find the solution for initial conditions $y_1(0) = 1$ and $y_2(0) = -1$, we can use the following technique:

```
t = sp.symbols("t", nonnegative=True)
y1, y2 = sp.symbols("y1, y2", cls=sp.Function, real=True)
odes = [y1(t).diff(t) + y1(t) - y2(t), y2(t).diff(t) + y2(t) - y1(t)]
ics = {y1(0): 1, y2(0): -1}
sol = sp.dsolve(odes, [y1(t), y2(t)], ics=ics)
print(sol)
```

```
[Eq(y1(t), exp(-2*t)), Eq(y2(t), -exp(-2*t))]
```

In engineering, it is common to express a set of differential equations as a state-space model, as in equation (4.24). The following example demonstrates how to solve these with SymPy.

Example 4.3

Consider the electromechanical schematic of a direct current (DC) motor shown in figure 4.6. A voltage source $V_S(t)$ provides power, the armature winding loses some energy to heat through a resistance R and stores some energy in a magnetic field due to its inductance L, which arises from its coiled structure. An electromechanical interaction through the magnetic field, shown as M, has torque constant K_M and induces a torque on the motor shaft, which is supported by bearings that lose some energy to heat via a damping coefficient B. The rotor's mass has rotational moment of inertia J, which stores kinetic energy. We denote the voltage across an element with v, the current through an element with i, the angular velocity across an element with Ω, and the torque through an element with T.

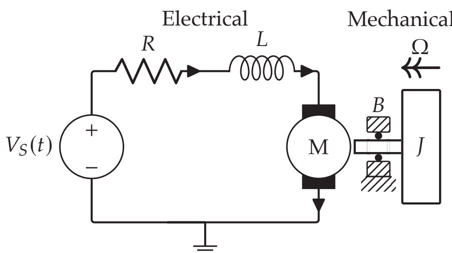


Figure 4.6. An electromechanical schematic of a DC motor.

A state-space model state equation in the form of equation (4.24a) can be derived for this system, with the result as follows:

$$\underbrace{\frac{d}{dt}\begin{bmatrix}\Omega_J \\ i_L\end{bmatrix}}_{\boldsymbol{x}'(t)} = \underbrace{\begin{bmatrix}-B/J & K_M/J \\ -K_M/L & -R/L\end{bmatrix}}_{A} \underbrace{\begin{bmatrix}\Omega_J \\ i_L\end{bmatrix}}_{\boldsymbol{x}(t)} + \underbrace{\begin{bmatrix}0 \\ 1/L\end{bmatrix}}_{B} \underbrace{\begin{bmatrix}V_S\end{bmatrix}}_{\boldsymbol{u}(t)} .$$

We choose $\boldsymbol{y} = \begin{bmatrix}\Omega_J\end{bmatrix}$ as the output vector, which yields output equation (i.e., equation (4.24b))

$$\underbrace{\begin{bmatrix}\Omega_J\end{bmatrix}}_{\boldsymbol{y}(t)} = \underbrace{\begin{bmatrix}1 & 0\end{bmatrix}}_{C} \underbrace{\begin{bmatrix}\Omega_J \\ i_L\end{bmatrix}}_{\boldsymbol{x}(t)} + \underbrace{\begin{bmatrix}0\end{bmatrix}}_{D} \underbrace{\begin{bmatrix}V_S\end{bmatrix}}_{\boldsymbol{u}(t)} .$$

Together, these equations are a state-space model for the system.
Solve the state equation for $\boldsymbol{x}(t)$ and the output equation for $\boldsymbol{y}(t)$ for the following case:

- The input voltage $V_S(t) = 1$ V for $t \geq 0$
- The initial condition is $\boldsymbol{x}(0) = \boldsymbol{0}$

We begin by defining the parameters and functions of time as SymPy symbolic variables and unspecified functions as follows:

```
R, L, K_M, B, J = sp.symbols("R, L, K_M, B, J", positive=True)
W_J, i_L, V_S = sp.symbols(
    "W_J, i_L, V_S", cls=sp.Function, real=True
)  # Ω_J, i_L, V_S
t = sp.symbols("t", real=True)
```

Now we can form the symbolic matrices and vectors:

```
A_ = sp.Matrix([[-B/J, K_M/J], [-K_M/L, -R/L]])  # A
B_ = sp.Matrix([[0], [1/L]])  # B
C_ = sp.Matrix([[1, 0]])  # C
D_ = sp.Matrix([[0]])  # D
x = sp.Matrix([[W_J(t)], [i_L(t)]])  # x
u = sp.Matrix([[V_S(t)]])  # u
y = sp.Matrix([[W_J(t)]])  # y
```

The input and initial conditions can be encoded as follows:

```
u_subs = {V_S(t): 1}
ics = {W_J(0): 0, i_L(0): 0}
```

The set of first-order ODEs comprising the state equation can be defined as follows:

```
odes = x.diff(t) - A_*x - B_*u
print(odes)
```

$$\begin{bmatrix} \frac{BW_J(t)}{J} + \frac{d}{dt}W_J(t) - \frac{K_M i_L(t)}{J} \\ \frac{K_M W_J(t)}{L} + \frac{d}{dt}i_L(t) + \frac{Ri_L(t)}{L} - \frac{V_S(t)}{L} \end{bmatrix}$$

```
x_sol = sp.dsolve(list(odes.subs(u_subs)), list(x), ics=ics)
```

The symbolic solutions for $x(t)$ are lengthy expressions. Instead of printing them, we will graph them for the following set of parameters:

```
params = {
    R: 1,  # (Ohms)
    L: 0.1e-6,  # (H)
    K_M: 7,  # (N·m/A)
    B: 0.1e-6,  # (N·m/(rad/s))
    J: 2e-6,  # (kg·m^2)
}
```

Create a numerically evaluable version of each function as follows:

```
W_J_ = sp.lambdify(t, x_sol[0].rhs.subs(params), modules="numpy")
i_L_ = sp.lambdify(t, x_sol[1].rhs.subs(params), modules="numpy")
```

Graph each solution as follows:

```
t_ = np.linspace(0, 0.000002, 201)
fig, axs = plt.subplots(2, sharex=True)
axs[0].plot(t_, W_J_(t_))
axs[1].plot(t_, i_L_(t_))
axs[1].set_xlabel("Time (s)")
axs[0].set_ylabel("$\\Omega_J(t)$ (rad/s)")
axs[1].set_ylabel("$i_L(t)$ (A)")
plt.show()
```

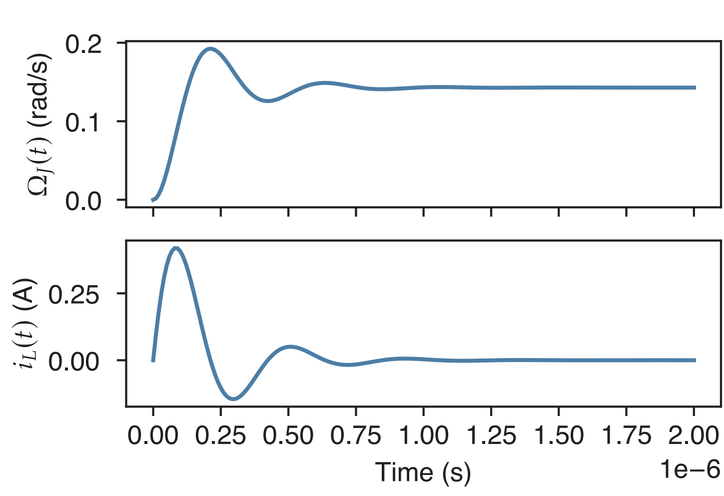


Figure 4.7. The state response to a unit step voltage input.

The output equation is trivial in this case, yielding only the state variable $\Omega_J(t)$, for which we have already solved. Therefore, we have completed the analysis.

4.8 Problems

Problem 4.1 5Z Let $s \in \mathbb{C}$. Use SymPy to perform a partial fraction expansion on the following expression:

$$\frac{(s+2)(s+10)}{s^4+8s^3+117s^2+610s+500}.$$

Problem 4.2 IE Let $x, a_1, a_2, a_3, a_4 \in \mathbb{R}$. Use SymPy to combine the cosine and sine terms that share arguments into single sinusoids with phase shifts in the following expression:

$$a_1 \sin(x) + a_2 \cos(x) + a_3 \sin(2x) + a_4 \cos(2x)$$

Problem 4.3 KR Consider the following equation, where $x \in \mathbb{C}$ and $a, b, c \in \mathbb{R}_+$,

$$ax^2 + bx + \frac{c}{x} + b^2 = 0.$$

Use SymPy to solve for x.

Problem 4.4 G9 Let $w, x, y, z \in \mathbb{R}$. Consider the following system of equations:

$$\begin{aligned} 8w - 6x + 5y + 4z &= -20 \\ 2y - 2z &= 10 \\ 2w - x + 4y + z &= 0 \\ w + 4x - 2y + 8z &= 4. \end{aligned}$$

Use SymPy to solve the system for w, x, y, and z.

Problem 4.5 IV Consider the truss shown in figure 4.8. Use a static analysis and the method of joints to develop a solution for the force in each member F_{AC}, F_{AD}, etc., and the reaction forces using the sign convention that tension is positive and compression is negative. The forces should be expressed in terms of the applied force f_D and the dimensions w and h only. Write a program that *solves for the forces symbolically* and answers the following questions:

a. Which members are in tension?
b. Which members are in compression?
c. Are there any members with 0 nominal force? If so, which?
d. Which member (or members) has (or have) the maximum compression?
e. Which member (or members) has (or have) the maximum tension?

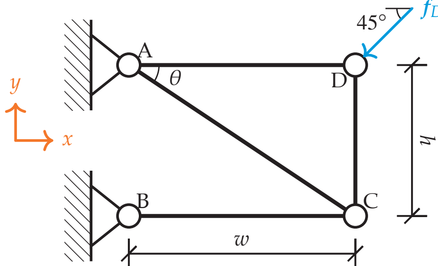


Figure 4.8. A truss with pinned joints, supported by two hinges, with an applied load f_D.

Problem 4.6 49 You are designing the truss structure shown in figure 4.9, which is to support the hanging of an external load $f_C = -f_C \hat{\boldsymbol{j}}$, where $f_C > 0$. Your organization plans to offer customers the following options:

- Any width (i.e., $2w$)
- A selection of maximum load magnitudes $L = f_C/\alpha \in \Gamma$, where $\Gamma = \{1\text{ kN}, 2\text{ kN}, 4\text{ kN}, 8\text{ kN}, 16\text{ kN}\}$, and where α is the factor of safety

As the designer, you are to develop a design curve for the dimension h versus half-width w for each maximum load $L \in \Gamma$, under the following design constraints:

- Minimize the dimension h
- The tension in all members is no more than a given T
- The compression in all members is no more than a given C
- The magnitude of the support force at pin A is no more than a given P_A
- The magnitude of the support force at pin D is no more than a given P_D

Use a static analysis and the method of joints to develop a solution for the force in each member F_{AB}, F_{AC}, etc., and the reaction forces using the sign convention that tension is positive and compression is negative. Create a Python function that returns h as a function of w for a given set of design parameters $\{T, C, P_A, P_D, \alpha, L\}$. Use the function to create a design curve h versus $2w$ for each $L \in \Gamma$, maximum tension $T = 81$ kN, maximum compression $C = 81$ kN, maximum support A load $P_A = 50$ kN, maximum support D load $P_D = 50$ kN, and a factor of safety of $\alpha = 5$.

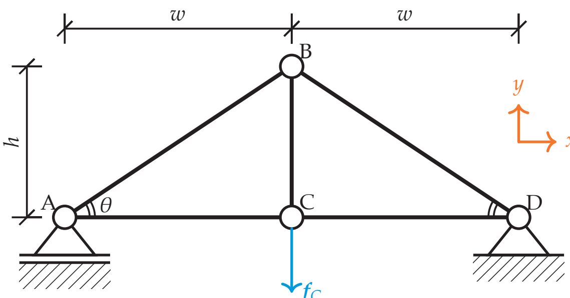


Figure 4.9. A truss with pinned joints, supported by a hinge and a floating support, with an applied load f_C.

Problem 4.7 W5 Consider an LTI system modeled by the state equation of the state-space model, equation (4.24a). A **steady state** of a system is defined as the state vector $\boldsymbol{x}(t)$ after the effects of initial conditions have become relatively small. For a constant input $\boldsymbol{u}(t) = \overline{\boldsymbol{u}}$, the constant state $\overline{\boldsymbol{x}}$ toward which the system's response decays can be found by setting the time derivative vector $\boldsymbol{x}'(t) = \mathbf{0}$.

Write a Python function `steady_state()` that accepts the following arguments:

- `A`: A symbolic matrix representing A
- `B`: A symbolic matrix representing B
- `u_const`: A symbolic vector representing $\overline{\boldsymbol{u}}$

The function should return `x_const`, a symbolic vector representing $\overline{\boldsymbol{x}}$.

The steady-state output converges to $\overline{\boldsymbol{y}}$ the corresponding output equation of the state-space model, equation (4.24b). Write a second Python function `steady_output()` that accepts the following arguments:

- `C`: A symbolic matrix representing C
- `D`: A symbolic matrix representing D
- `u_const`: A symbolic vector representing $\overline{\boldsymbol{u}}$
- `x_const`: A symbolic vector representing $\overline{\boldsymbol{x}}$

This function should return `y_const`, a symbolic vector representing $\overline{\boldsymbol{y}}$.

Apply `steady_state()` and `steady_output()` to the state-space model of the circuit shown in figure 4.10, which includes a resistor with resistance R, an inductor with inductance L, and capacitor with capacitance C. The LTI system is represented by equation (4.24) with state, input, and output vectors

$$\boldsymbol{x}(t) = \begin{bmatrix} v_C(t) \\ i_L(t) \end{bmatrix}, \ \boldsymbol{u}(t) = \begin{bmatrix} V_S \end{bmatrix}, \ \boldsymbol{y}(t) = \begin{bmatrix} v_C(t) \\ v_L(t) \end{bmatrix}$$

and the following matrices:

$$A = \begin{bmatrix} 0 & 1/C \\ -1/L & -R/L \end{bmatrix}, \; B = \begin{bmatrix} 0 \\ 1/L \end{bmatrix}, \; C = \begin{bmatrix} 1 & 0 \\ -1 & -R \end{bmatrix}, \; D = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Furthermore, let the constant input vector be

$$\overline{\boldsymbol{u}} = \begin{bmatrix} \overline{V_S} \end{bmatrix},$$

for constant $\overline{V_S}$.

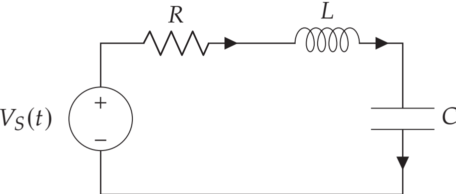


Figure 4.10. An RLC circuit with a voltage source $V_S(t)$.

Problem 4.8 8U Consider the electromechanical state-space model described in example 4.3. For a given set of parameters, input voltage, and initial conditions, the following vector-valued functions have been derived:

$$\boldsymbol{F} = \begin{bmatrix} \int_0^t v_R(t)\,dt \\ \int_0^t v_L(t)\,dt \\ \int_0^t \Omega_B(t)\,dt \\ \int_0^t \Omega_J(t)\,dt \end{bmatrix} = \begin{bmatrix} \exp(-t) \\ \exp(-t) \\ 1-\exp(-t) \\ 1-\exp(-t) \end{bmatrix}, \quad \boldsymbol{G} = \begin{bmatrix} \int_0^t i_R(t)\,dt \\ \int_0^t i_L(t)\,dt \\ \int_0^t T_B(t)\,dt \\ \int_0^t T_J(t)\,dt \end{bmatrix} = \begin{bmatrix} \exp(-t) \\ \exp(-t) \\ 1-\exp(-t) \\ \exp(-t) \end{bmatrix}$$

The instantaneous power lossed or stored by each element is given by the following vector of products:

$$\boldsymbol{\mathcal{P}}(t) = \begin{bmatrix} v_R(t) i_R(t) \\ v_L(t) i_L(t) \\ \Omega_B(t) T_B(t) \\ \Omega_J(t) T_J(t) \end{bmatrix}.$$

The energy $\mathcal{E}(t)$ of the elements, then, is

$$\boldsymbol{\mathcal{E}}(t) = \int_0^t \boldsymbol{\mathcal{P}}(t) dt.$$

Write a program that satisfies the following requirements:

a. It defines a function `power(F, G)` that returns the symbolic power vector $\boldsymbol{\mathcal{P}}(t)$ from any inputs $\boldsymbol{F}$ and $\boldsymbol{G}$

b. It defines a function `energy(F, G)` that returns the symbolic energy $\mathcal{E}(t)$ from any inputs $\boldsymbol{F}$ and $\boldsymbol{G}$ (`energy()` should call `power()`)
c. It tests the `energy()` on the specific $\boldsymbol{F}$ and $\boldsymbol{G}$ given above

Problem 4.9 FJ For the circuit and state-space model given in problem 4.7, use SymPy to solve for $\boldsymbol{x}(t)$ and $\boldsymbol{y}(t)$ given the following:

- A constant input voltage $V_S(t) = \overline{V_S}$
- Initial condition $\boldsymbol{x}(0) = \boldsymbol{0}$

Substitute the following parameters into the solution for $\boldsymbol{y}(t)$ and create numerically evaluable functions of time for each variable in $\boldsymbol{y}(t)$:

$$R = 50\ \Omega,\ L = 10 \cdot 10^{-6}\ \text{H},\ C = 1 \cdot 10^{-9}\ \text{F},\ \overline{V_S} = 10\ \text{V}.$$

Plot the outputs in $\boldsymbol{y}(t)$ as functions of time, making sure to choose a range of time over which the response is best presented. *Hint:* An appropriate amount of time is on the scale of microseconds.

5 Numerical Analysis I: Techniques

asdf

5.1 Problems

MZ

Problem 5.1 M2 asdf

Notebooks

B Documenting and Presenting Programs

asdf

Version Control

D Lists of Figures and Tables

D.1 List of Figures

QJ

1.1 The Spyder IDE when it first loads 4

2.1 The functional design method (a) at the highest level and (b) in general, for any level. 47

3.1 A “computer room” at the NACA (precursor to NASA) high-speed flight station in 1949 (NASA 2002). 55

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). 74

3.3 A graph of polynomial $f(x)$. 76

3.4 Ideal gas pressure versus volume for different temperatures. 78

3.5 A bar chart of thermal conductivity for metals (data from Carvill (1994)). 80

3.6 A histogram of my movie ratings on a 0–10 scale. 81

3.7 A polygon and vectors from R to two consecutive vertices. 88

4.1 A symbolic expression tree for `sp.sqrt(3)/2`. 93

4.2 A truss with pinned joints, supported by a hinge and a floating support, with an applied force f_F. 110

4.3 A graph of $\sqrt{r^2+1}$, where $r = w/h$. 114

4.4 A resistor circuit design for example 4.2. 119

4.5 A design graph for resistors R_1, R_2, and R_3. 122

4.6 An electromechanical schematic of a DC motor. 133

4.7 The state response to a unit step voltage input. 135

4.8 A truss with pinned joints, supported by two hinges, with an applied load f_D. 137

4.9 A truss with pinned joints, supported by a hinge and a floating support, with an applied load f_C. 138

4.10 An RLC circuit with a voltage source $V_S(t)$. 139

D.2 List of Tables

9L

1.1 Boolean and comparison operators on Boolean and integer inputs `x` and `y` 9
1.2 Format specifier terms. 11
1.3 Format specifier types. 12
1.4 Some particularly useful string methods. 12
1.5 Mutability of commonly used built-in types. 14
1.6 Commonly used list methods for a list `l`. 15
1.7 Dictionary instance methods for dictionary instance `d` and class method for class `dict`. 18
2.1 Python standard library modules of particular interest to the engineer. 34
3.1 JSON to Python reading conversion. 69
3.2 Python to JSON writing conversion. 70
4.1 Elementary mathematical functions in SymPy. 95

Bibliography

Abelson, Hal, and Gerald Jay Sussman. 2016. *Structure and Interpretation of Computer Programs.* 2nd ed. MIT Press (orig. 1996). https://engineering-computing.ricopic.one/5n.

Carvill, James. 1994. *Mechanical Engineer's Data Handbook.* Butterworth-Heinemann.

Cross, Nigel. 2021. *Engineering Design Methods: Strategies for Product Design, 5th Edition.* 5th ed. Wiley. http://gen.lib.rus.ec/book/index.php?md5=988D6046DBEE3E1452E2F099079B445E.

Filik, Ruth, Alexandra Ţurcan, Christina Ralph-Nearman, and Alain Pitiot. 2019. "What is the difference between irony and sarcasm? An fMRI study." [in eng]. *Cortex* 115 (June): 112–122. https://doi.org/10.1016/j.cortex.2019.01.025. https://engineering-computing.ricopic.one/e8.

Gilens, Martin, and Benjamin I. Page. 2014. "Testing Theories of American Politics: Elites, Interest Groups, and Average Citizens." *Perspectives on Politics* 12 (3): 564–581. https://doi.org/10.1017/S1537592714001595.

Gonzalez, Ryan, Philip House, Ivan Levkivskyi, et al. 2024. *PEP 526 – Syntax for Variable Annotations,* February (orig. 2016). https://engineering-computing.ricopic.one/9x.

Google. 2024. *Google Python Style Guide,* February. https://engineering-computing.ricopic.one/ne.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, et al. 2020. "Array Programming with NumPy." *Nature* 585, no. 7825 (September): 357–362. https://doi.org/10.1038/s41586-020-2649-2. https://doi.org/10.1038/s41586-020-2649-2.

Hunt, A., and D. Thomas. 1999. *The Pragmatic Programmer: From Journeyman to Master.* Pearson Education.

Hunter, J. D. 2007. "Matplotlib: A 2D graphics environment." *Computing in Science & Engineering* 9 (3): 90–95. https://doi.org/10.1109/MCSE.2007.55.

Johnston, Nathaniel, and Dave Greene. 2022. *Conway's Game of Life: Mathematics and Construction.* Self-published. https://doi.org/10.5281/zenodo.6097284.

Kreyszig, E. 2010. *Advanced Engineering Mathematics.* 10th ed. John Wiley & Sons.

Langa, Łukasz, and contributors to Black. 2024. *Black: The Uncompromising Python Code Formatter,* February. https://engineering-computing.ricopic.one/8n.

NASA. 2002. *NACA High Speed Flight Station "Computer Room",* June (orig. 1949). https://engineering-computing.ricopic.one/mz.

NumPy Developers. 2024a. *NumPy Reference,* February (orig. 2022). https://engineering-computing.ricopic.one/u5.

NumPy Developers. 2024b. *NumPy User Guide,* February (orig. 2022). https://engineering-computing.ricopic.one/5y.

NumPy Developers. 2024c. *NumPy: The Absolute Basics for Beginners,* February (orig. 2022). https://engineering-computing.ricopic.one/3l.

Python Community. 2024a. *Python 3.X Documentation,* January. https://engineering-computing.ricopic.one/n3.

Python Community. 2024b. *Python Package Index,* January. https://engineering-computing.ricopic.one/e0.

Python Community. 2024c. *Python Packaging User Guide,* January. https://engineering-computing.ricopic.one/w9.

Rossum, Guido van, Jukka Lehtosalo, and Łukasz Langa. 2024. *PEP 484 – Type Hints,* February (orig. 2014). https://engineering-computing.ricopic.one/z9.

Rossum, Guido van, Barry Warsaw, and Alyssa Coghlan. 2024. *PEP 8 – Style Guide for Python Code,* February (orig. 2001). https://engineering-computing.ricopic.one/3z.

SymPy Development Team. 2023a. *Advanced Expression Manipulation,* May. https://engineering-computing.ricopic.one/95.

SymPy Development Team. 2023b. *Core [SymPy Documentation],* May. https://engineering-computing.ricopic.one/92.

SymPy Development Team. 2023c. *Simplification,* May. https://engineering-computing.ricopic.one/14.

SymPy Development Team. 2023d. *Writing Custom Functions,* May. https://engineering-computing.ricopic.one/3c.

Tufte, Edward R. 2001. *The Visual Display of Quantitative Information.* 2nd ed. Graphics Press.

Yasskin, Jeffrey. 2024. *PEP 3141 – A Type Hierarchy for Numbers,* February (orig. 2007). https://engineering-computing.ricopic.one/41.

Contributors

Associate Professor Rico A.R. Picone
Department of Mechanical Engineering
Saint Martin’s University
Lacey, Washington, USA