Introduction to Computational Physics Lecture Notes

James Farrell and Jure Dobnikar

Spring 2023

Contents

Part I

TOOLS

Chapter 1

Programming with Python

1.1 Introduction

The first chapter of this part is a brief introduction to programming in the Python language. Without an understanding of how to program computers, the rest of the information in this book is quite useless! We encourage you to take this chapter very seriously. Whether you are completely new to programming, or have a lot of experience, you should find something of value here. The skills you will begin to learn in this chapter have applications not only in physics, but also in science more generally, and in other fields besides.

Many books, introductory and advanced, have been written on this topic. We will take a pragmatic approach, focusing on the ideas that will be crucial to completing the exercises in this course. For an introduction to Python programming set in a broader context, we encourage you to read A Byte of Python, a free, online tutorial aimed at total beginners, which is also available in a Mandarin translation.

1.1.1 What is Python?

Python is a programming language—a way of telling a computer what to do. In particular, Python is an **interpreted**, **object-oriented**, **high-level** programming language.

An **interpreted** language is a programming language whose programs are translated into machine code by an interpreter at run-time—this is opposed to a **compiled** language, whose programs are translated into machine code *before* run-time.

Object-oriented programming is a **programming paradigm** based on the concept of "objects", which can contain data, in the form of fields (often known as attributes), and code, in the form of procedures (often known as methods). Actually, programs written in Python are not restricted to the object-oriented paradigm—Python supports **many programming paradigms**, such as object-oriented, imperative, procedural, and functional programming, to greater or lesser degree. More information on programming paradigms can be found here.

A **high-level** language is one with strong abstraction from the details of the computer. Programs written in high-level languages are usually easier to understand and modify than their low-level counterparts, and hide from the programmer the fine details of how computers actually work (the details are abstracted away), such as memory management. Programs written in **low-level** languages also have advantages; for example, the programmer can write programs tailored to the specific machine on which the programs are to be run, resulting in more efficient code.

As such, Python is a great first language for a new programmer: she need not worry about the details of compilation, as her programs are **interpreted** when run; she may explore many **programming paradigms**, and learn which paradigm is appropriate for which task; and from the very beginning she may write programs using using complex structures, written with a syntax similar to natural language, owing to Python being a **high-level** language. However, her programs will not be as efficient as those written in other languages, whose programs interface directly with hardware (**low-level languages**), or can be optimized at compilation time (**compiled languages**).

1.1.2 Installing Python

In order to complete this course, you will need two things: a working Python installation; an integrated development environment (IDE). If you understand all of the terms in the preceding sentence, you are probably ready to begin. If not, I recommend you use the `Conda` package and environment management system and the `PyCharm` IDE.

`Conda`

Instructions on how to install `Conda` on Windows, Mac, or Ubuntu. Choose the Python 3 version.

`PyCharm`

Instructions on how to install `PyCharm`.

1.2 Basics I

By the end of this section, you should be able to understand and write python code that looks like the following:

```
def factorial(n: int) -> int:  # function definition statement
    """

    evaluates n! = n * (n - 1) * ... * 2 * 1
    0! evaluates to 1

    >>> factorial(0)
    1

    >>> factorial(10)
    3628800

    >>> factorial(-1)
    Traceback (most recent call last):
    ValueError: n! is undefined for n less than zero

    >>> factorial(3.141)
    Traceback (most recent call last):
    TypeError: n is not an integer

    :param n: element of the factorial sequence to be evaluated
    :return: n!
    """

    if n < 0:  # if statement
        raise ValueError("n! is undefined for n less than zero")  # raise statement
    elif not isinstance(n, int):
        raise TypeError("n is not an integer")  # raise statement

    n_factorial = 1  # assignment statement

    while n > 1:  # while statement
        n_factorial = n_factorial * n  # assignment statement
        n = n - 1  # assignment statement

    return n_factorial  # return statement
```

As you might have guessed, this is program that evaluates the factorial of a number. We'll go through every element one step at a time.

Tip The code elements `## function definition statement`, `## if statement`, etc are *comments*. Comments are ignored when the code is run, but provide useful information about the code. You should use comments liberally! In the python language, anything following a `##` that is not part of a *string literal* is parsed as a comment. Generally, comments should be more informative than these examples; for example, it is quite obvious (to the initiated) that line 1 is a function definition statement. Comments should explain your intent, not simply reproduce what is obvious from the code.

The core functionality is contained in lines 32–36:

```
32      while n > 1:  # while statement
33          n_factorial = n_factorial * n  # assignment statement
34          n = n - 1  # assignment statement
35
36      return n_factorial  # return statement
```

Let's take a closer look at line 32.

1.2.1 Assignment statements, names, expressions

Line 32 is an *assignment statement*. *Statements* are sections of code that *do* something. Assignment statements are the most fundamental statements in the language - they allow you to store the result of a calculation in a variable.

On the left-hand-side (LHS) of the assignment statement, there should be a *name*, or *identifier*. In this instance, a name is what you might refer to as a variable. In our example, the name is `n_factorial`. There are some rules that determine whether a name is valid. The left-hand-side of an assignment statement must be a *valid name*.

- names can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
- names like `myClass`, `var_1` and `print_this_to_screen`, all are valid.
- a name cannot start with a digit.
- `1variable` is invalid, but `variable1` is perfectly fine.
- it is unwise to use the names of built-in functions (`print`, `list`, `input`, etc) as names.
- it is **illegal** to use *reserved words*, or *keywords* (`False`, `for`, `return`, etc) as names.

Tip Names should be chosen to maximise code readability. Avoid single letter names (e.g. `i, j, x, y`) where a more informative choice can be made (e.g. `height, weight, number_of_people`). Follow the conventions in the style guide.

- function and variable names should be `lower_case_with_underscores`
- class names should be `CamelCase`
- names for constants should be `UPPER_CASE_WITH_UNDERSCORES`
- module names should be short and all lowercase

The right-hand-side (RHS) of the assignment statement must be a *valid expression*. Expressions can be made up of *literals*, *names*, *operators*, and *function calls*.

- `1` and `"hello world"` are literals, and valid expressions
- `2+3` and `print(2+3)` are valid expressions
- `1+`, `/2`, `"string` are not expressions and will cause a syntax error
- "syntax" errors, errors in *grammar*, occur when you have written something that the interpreter cannot understand, e.g., an invalid expression,

```
>>> 1 +
  File "<stdin>", line 1
    1 +
      ^
SyntaxError: invalid syntax
```

(The code block above is a python console session. Lines beginning with `>>>` are typed by you; lines without are the console output.)

When the assignment statement is executed,

- the expression on the RHS is evaluated
- the name on the LHS is *bound* to the result

The name then behaves like that result in subsequent expressions and statements.

```
>>> n_factorial = 1  # assignment statement
>>> n_factoral  # expression statement
1
```

Tip In the previous console session, the second line is an *expression statement*. This a statement made that is just an expression. If you type an expression statement into a console session, and the expression evaluates to something other than `None`, the result will be printed to the screen. This only works in console sessions; an expression statement in a `\*.py` file will be executed, but it's value will **not** be printed to the screen.

1.2.2 Literals, types, operators

The simplest expressions are *literals*. Literals are notations for constant values of some built-in *type*. The type of an object determines the result of applying *operators* to it.

```
>>> 1 + 2
3
```

In the above console session, `1` and `2` are *integer literals*. Their type is `int`, and their values are the integers 1 and 2. `+` is a *binary operator*. When the types of the objects on the LHS and RHS of the `+` are `int`, integer addition will be performed.

Arithmetic operators

The arithmetic operators are

- `+` performs addition
- `-` performs subtraction
- `*` performs multplication
- `//` performs integer division
- `/` performs float division
- `%` performs modulo
- `**` performs exponentiation
- `@` performs matrix multiplication (Python 3.6+)

As a *unary operator*, `-` in the expression `-x` multiplies `x` by -1. For all of these operators except float division and exponentiation, if the LHS and RHS are both `int`, the expression `LHS ∘ RHS` evaluates to an `int`.

Numeric types

As well as the `int` type for representing integers, there are the `float` type, for representing floating point numbers (real numbers, with some caveats) and the `complex` type, for representing complex numbers. Floats can be entered in decimal or exponential format. Imaginary numbers are entered the same way, but with a letter `j` at the end. Complex numbers are entered as a sum of a float and an imaginary number.

```
 >>> 1000.0 # float
1000.0
>>> 1e3
1000.0
>>> 1e20
1e+20
>>> 1e-20
1e-20
>>> 2.5j  # complex
2.5j
>>> 1e10 + 1e-10j  # complex
(10000000000+1e-10j)
```

When the LHS and RHS are not the same numeric type, the following *arithmetic conversion* rules are observed:

- if either argument is a complex number, the other is converted to complex
- otherwise, if either argument is a float, the other is converted to float
- otherwise, both must be integers and no conversion is necessary

What you might consider a fourth numeric type is the `bool` type for boolean numbers. A `bool` is equal to either `True` or `False`. In arithmetic expressions, `bool` objects are treated as 0 or 1, for `False` and `True`, respectively. The type of a literal, name, or other object can be discovered by using the `type` function,

```
>>> type(True)
<class 'bool'>
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type(1.0j)
<class 'complex'>
```

Tip Since Python 3, the `/` operator performs float division, regardless of whether both the LHS and RHS are `int`. Take note of the following examples.

```
>>> 1 // 2
0
>>> 1 / 2
0.5
>>> 2 / 1
2.0
```

Time to go back to our `factorial` code.

```
33          n_factorial = n_factorial * n  # assignment statement
34          n = n - 1  # assignment statement
```

Lines 33 and 34 are assignment statements with arithmetic expressions on the RHS. You will notice in line 33 that `n_factorial` appears on both sides of the `=` sign. This is not a problem, as the RHS is evaluated first, then the LHS is about to that value. In variable terms, the old value of `n_factorial` is replaced by `n_factorial * n`.

To understand line 32, we have to familiarise ourselves with *comparison operators*, and *while statements*—a kind of *control flow*.

Comparison operators

As well as arithmetic operators, the Python language uses *comparison operators*. Comparison operators compare two expressions, and evaluate to either `True` or `False`. Among the comparison operators are

- `==` evaluates to `True` if the LHS and RHS are equal, `False` otherwise
- `!=` is the negation of `==`
- `<` and `<=` evaluate to `True` if the LHS is less than, or less than or equal to the RHS
- `>` and `>=` evaluate to `True` if the LHS is greater than, or greater than or equal to the RHS
- `is` evaluates to `True` if the LHS and RHS are *identical objects*, `False` otherwise
- `is not` is the negation of `is`

Info: The distinction between `is` (identical) and `==` (equal) is a subtle one. Consider the following examples.

```
>>> 1 == 1.0  # 1 is converted to a float
True
>>> 1 is 1.0  # 1 is an int literal, 1.0 is a float literal
False
>>> x = 1
>>> y = 1
>>> x == y  # x and y are both equal to 1
True
>>> x is y  # x and y are both int with value 1
True
```

Besides arithmetic and comparison operators, there are also boolean operators (for use with `bool` arguments) and bitwise operators (for use with `int` arguments).

```
32      while n > 1:  # while statement
```

In line 32 of our `factorial` code, the expression `n > 1` evaluates to `True` if `n` is greater than `1`, and `False` otherwise.

1.2.3 The `while` statement (indefinite iteration)

In many situations, we want the behaviour of our program to depend on the current properties of a variable, or that of some expression containing it. For example, we may want to *loop* over a set of statements until some condition obtains; or perhaps we would like our program to treat negative integers and positive integers differently. We can achieve these aims by controlling the flow of execution of our program with *control flow statements*.

One such control flow statement is the `while` statement. The `while` statement has the form

```
1 while expr:
2     suite
3 other code
```

When a `while` statement is encountered, the expression `expr` is evaluated. If `expr` evaluates to an object that is considered *false*, then line 2 is ignored, and execution moves to line 3. If `expr` evaluates to an object that is considered *true*, then the set of statements in the block `suite` is executed. After executing `suite`, the *truth value* of `expr` is tested again. This *loop* continues until `expr` evaluates to a *false* object.

Notice the *indentation* on line 2. While lines 1 and 3 begin at the same horizontal position, line 2 is *indented* by 4 spaces. The content of the suite is defined by those statements immediately following the `while` statement at a greater level of indentation. When a statement with the same indentation or lesser is encountered, it is not executed, and execution returns to the `while` statement.

Consider the following console session that calculates the sum of the integers less than or equal to 5.

```
>>> total = 0
>>> n = 5
>>>
>>> while n > 0:
...     total = total + n
...     n = n - 1
...
>>> total
15
>>> n
0
```

When we first encounter line four, the value of `n` is `5`. `5 > 0` evaluates to `True`, which is a true value, so the statements on lines five and six are executed. The value of `total` is updated to `5` (line five), and the value of `n` is updated to `4` (line six). We then return to line four. `4 > 0` still evaluates to a true value, so lines five and six are executed once more. After lines five and six have been executed a total of five times, the value of `total` is 15, and the value of `n` is `0`. `0 > 0` evaluates to `False`, which is a false value, so execution moves to line eight, and we escape the loop.

What is a true or false object? For the purposes of truth-value testing,

- the constants `None` and `False` are defined to be false
- zero of any numeric type is defined to be false: `0, 0.0, 0.0e10, 0j, Decimal(0), Fraction(0, 1)`
- empty sequences and collections are also false: `'', (,), [], {}, set(), range(0)`

As a consequence, `while True:`, `while 1:`, and `while 3.142:` are functionally equivalent.

Returning to our initial problem,

```
33          n_factorial = n_factorial * n  # assignment statement
34          n = n - 1  # assignment statement
```

we can see that lines 33 and 34 will be executed a total of `n - 1` times, with the variable `n_factorial` multiplied by every integer `m` for `1 < m < n`.

Question 1 The Fibonacci Sequence: part 1

You are likely familiar with the Fibonacci sequence:

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0;\ F_1 = 1$$

In the limit of large n, the ratio F_n/F_{n-1} tends to the golden ratio, ϕ:

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \phi = \frac{1+\sqrt{5}}{2}$$

Using only the Python we have learned so far, [a] find the smallest n such that:

$$\left\| \frac{F_n}{F_{n-1}} - \phi \right\| < 10^{-10}$$

[a] you may also need to use the built-in `abs` function; `abs(x)` returns the absolute value of `x`.

1.2.4 Functions and modules

Consider the following console session.

```
>>> print(888)
888
```

`print` is a *function*. It's purpose is to take it's *argument* (`888`) and print it to the screen. We pass the argument to the function *via* the *function call operator*, `()`.

- generally speaking, functions can take any number of arguments (including none)
- functions *return* some object (the result of the operation of the function on its arguments)
- if an expression is a function call, it evaluates to the function's return value
- the return value of `print` is always `None`

Python has many built-in functions, such as `print`, that are always available. Other functions can be *imported* from *modules*. For mathematical functions other than basic arithmetic (i.e., `+, -, *, /, **`), we can import the math module with an *import statement*,

```
>>> import math # import statement
>>> math.sqrt(4.0) # square root of 4
2.0
>>> math.sqrt(2.0) # square root of 2
1.4142135623730951
>>> math.acos(1.0) # arccos of 1
0.0
```

The notation `module.object` refers to the attribute `object` of `module`. The attribute could be a constant, a function, a class, or another module. In the above examples, we access the `sqrt` function of the `math` module. As well as importing functions from `math`, we can import constants,

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
```

Having to retype `math` every time is tedious, and makes the code less readable. Instead, we can import math with an *alias*,

```
>>> import math as m
>>> m.asin(1) # arcsin of 1
1.5707963267948966
```

or import only the *names* we want to use,

```
>>> from math import atan
>>> atan(1.0) # arctan of 1
0.7853981633974483
```

The `def` and `return` statements

We can define our own functions. A function definition has the following form:

```
1 def function_name(argument_list):  # function definition statement
2     suite  # function body
3     return return_list  # return statement, optional
```

Unlike a `while` statement, when the `def` statement is encounted, the suite is not executed. Rather, the suite is checked for syntax errors, and then a *function object* is created. When the function is called with the function call operator, *i.e.* `function_name()`, the statements in `suite` are executed, and `function_name()` evaluates to `return_list`. If `return_list` is not provided, or the `return statement` is omitted altogether, the function call evaluates to `None`.

The `suite` is defined by the continuous block of statements following the `def` statement that are at a greater level of indentation that the `def` statement itself (just like we saw for the `suite` in the `while` statement).

We can import our own functions in the same way we import functions from modules.

```
# square.py

def square(x: float) -> float:
    return x * x
```

```
# script.py

from square import square

x = 4
x_squared = square(x)
print(x)
```

Functions can be written with two types of arguments:

- *positional* arguments, that have no default value, and must by explicitly provided to a function call
- *keyword* arguments, that have some default value, and may be omitted from a function call

Consider a function that takes two numbers, x and n, and returns the n^{th} power of x. We give n a default value 1. The correct way to construct the `def` statement is as in the following:

```
# power.py

def power(x: float, n: int=1):
return x ** n
```

Positional arguments are specified first, followed by keyword arguments. The `def` statement

```
def power(n=1, x):
    return x ** n
```

is a syntax error.

The rules for function calls are somewhat more lax,

```
>>> from power import power
>>>
>>> a, b = 2, 3
>>> power(x=a)  # n can be omitted from the argument list
2
>>>
>>> power(n=a) # x cannot be omitted from the argument list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'x'
>>>
>>> power(x=a, n=b) # x can be supplied as a keyword argument
8
```

```
>>>
>>> power(n=a, x=b) # if all arguments are supplied as keyword arguments, the order is arbitrary
9
>>>
>>> power(a, b) # if all arguments are supplied as positional arguments,
8
>>>
>>> power(b, a) # the order is critical
9
>>>
>>> power(n=a, x) # positional arguments cannot follow keyword arguments
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

We recommend you follow the pattern in lines 3 and 9 to maximise the readability of your code and avoid errors.

Type hints

In statically-typed languages, such as FORTRAN and C, the types of objects and arguments to functions are fixed as or before they are used. Python is by contrast a dynamically-typed language; functions in Python do not require arguments of particular types, and the types of arguments are not checked by the interpreter at run time. Additionally, an identifier may refer to objects of different types over the course of its lifetime. Nevertheless, it is useful to annotate Python code with *type hints*, indicating the types or behaviours that are expected in a particular situation. In the function definition statement of the `factorial` function,

```
1 def factorial(n: int) -> int:  # function definition statement
```

type hints are used to indicate that both the type of the argument `n`, and the return type are both `int`. Even with type hints, the types of arguments are never checked unless we explicitly require it (*e.g.,* on line 27 of the same code), but provide useful information to the user and the IDE, as well as appearing in the output of the `help` function.

docstrings and doctests

The syntax of lines 1 and 36 in `factorial` are now clear to us. We now turn to the largest section of the code. 22 of the 36 lines, from lines 2 to 23, are devoted to *documentation*. Inline comments (`## like this`) help to elucidate small blocks of statements, but *docstrings* draw the bigger picture.

```
2     """
3
4     evaluates n! = n * (n - 1) * ... * 2 * 1
5     0! evaluates to 1
6
7     >>> factorial(0)
8     1
9
10    >>> factorial(10)
11    3628800
12
13    >>> factorial(-1)
14    Traceback (most recent call last):
15    ValueError: n! is undefined for n less than zero
16
17    >>> factorial(3.141)
18    Traceback (most recent call last):
19    TypeError: n is not an integer
20
21    :param n: element of the factorial sequence to be evaluated
```

```
22      :return: n!
23      """
```

A docstring immediately follows a `def` statement and is enclosed in triple double quotes. By convention, the first few lines (here, lines 4–5) describe in words the problem that the function solves. It should provide a sufficiently detailed explanation for a user to know whether it is the function they are looking for.

The following block of lines contain usage examples in the form of transcribed console sessions (lines 7–19). These lines a more than just hints—we will come back to them shortly.

The last block of lines should contain details about the parameters and return value (lines 21–22). From line 1, we can see the author intended for the `factorial` function to take an integer parameter, `n`. This comment does not imply if you give the function a non-integer value then an error will be thrown (lines 17–19, however, state this explicitly); rather it implies that problem-free execution is not *guaranteed*. An explicit error is really the best-case scenario; in the worst-case scenario, a function returns a nonsense value, execution proceeds silently, and the program returns an incorrect result *that you may not even be able to identify as such!*

A function's docstring is stored in its `__doc__` attribute.

```
>>> from factorial import factorial
>>> print(factorial.__doc__)

    evaluates n! = n * (n - 1) * ... * 2 * 1
    0! evaluates to 1

    >>> factorial(0)
    1

    >>> factorial(10)
    3628800

    >>> factorial(-1)
    Traceback (most recent call last):
    ValueError: n! is undefined for n less than zero

    >>> factorial(3.141)
    Traceback (most recent call last):
    TypeError: n is not an integer

    :param n: element of the factorial sequence to be evaluated

    :return: n!

>>>
```

In a console session, the `help` function, when supplied with an object as its argument, will bring up a page of useful information of which the docstring forms part.

Now back to those usage examples (lines 7–19); in addition to being handy hints, they form a suite of tests. The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. If your tests fail, then either

a. your docstring needs to be updated, or
b. your code no longer works properly.

`doctest` can be invoked in a terminal session,

```
farrelljd@sommerfugl:~/2019/programming_course$ python3 -m doctest factorial.py
farrelljd@sommerfugl:~/2019/programming_course$
farrelljd@sommerfugl:~/2019/programming_course$ python3 -m doctest -v factorial.py
Trying:
    factorial(0)
Expecting:
```

```
    1
ok
Trying:
    factorial(10)
Expecting:
    3628800
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
    ValueError: n! is undefined for n less than zero
ok
Trying:
    factorial(3.141)
Expecting:
    Traceback (most recent call last):
    TypeError: n is not an integer
ok
1 items had no tests:
    factorial
1 items passed all tests:
   4 tests in factorial.factorial
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
farrelljd@sommerfugl:~/2019/programming_course$
```

where the `-v` flag enables verbose output. In a PyCharm session, right-click on your code and choose *Run 'Doctest <module>'*.

Writing good documentation may seem like a hassle now, but it saves a lot of time in the long run. Best practice is to write a docstring and doctests *before* you implement your function! That way, you have a clear statement of how your function should behave in a variety of circumstances, and can periodically check your progress with the doctest module.

1.2.5 The `if` statement

`while` statements allow us to repeatedly execute blocks of code while some condition is satisfied. `if` statements allow use to execute different blocks of code depending on the truth value(s) of a set of expressions.

The `if` statement has the form:

```
1  if expression1:
2      suite1
3  elif expression2:
4      suite2
5  ...
6  elif expressionY:
7      suiteY
8  else:
9      suiteZ
10 other code
```

The flow is simple: if `expression1` evaluates to a true value, `suite1` is executed, and the interpreter moves to the end of the `if` block (line 10). Otherwise, the truth value of `expression2` is evaluated; if it is true, `suite2` is executed, and the interpreter moves to the end of the `if` block. All expressions are evaluated until a true value is obtained, whereupon the corresponding suite is executed, and all subsequent expressions ignored. If no expression evaluates to a true value, the suite following the `else` statement is executed.

An `if` block has exactly one `if` statement, zero or more `elif` statements, and zero or one `else` statement.

continue and break

Complex control flow can be implemented by combining `if` and `while` statements. Two useful statements for use within `while` (and `for`) blocks are `continue` and `break`. When a `continue` statement is encountered in a suite, the remainder of the suite is skipped, and execution resumes at the `while` statement. When a `break` statement is encountered in a suite, the remainder of the suite is skipped, and execution resumes *at the end* of the `while` block. The `else` clause of the `while` block, if present, **is not executed**.

```
>>> i = 0
>>> while True:  # loop forever
...     i += 1
...     if i % 2 == 0:  # skip even numbers
...             continue
...     elif i % 5 == 0:  # --> print odd numbers less than five
...             break
...     print(i)
... else:
...     print("this line is never executed")
...
1
3
>>>
```

Returning to the factorial code,

```
25      if n < 0:  # if statement
26          raise ValueError("n! is undefined for n less than zero")  # raise statement
27      elif not isinstance(n, int):
28          raise TypeError("n is not an integer")  # raise statement
```

line 25 tests whether `n` is less than zero. If true, line 26 is executed. Otherwise, line 27 tests whether `n` is an not instance of the type `int` by means of the built-in `isinstance` function. If true, line 28 is executed.

1.2.6 Exception handling

What should we do if the value supplied to our `factorial` function is less than zero? If we do nothing, the `while` suite is skipped, and, thanks to line 30, the return value for all `n < 0` is 1.

This is a problem. Somewhere, some piece of code has called the `factorial` function with a negative number as its argument, and nobody noticed. In all likelihood, that number is negative because of *an error somewhere else in the code*. We would like to find that error *as soon as possible*. We can do this by *raising an exception*. An exception is not a Python error, but something we should *take exception to*, *i.e.* an outcome we should reject.

A `raise` statement has the form

```
1  raise exception_class("error_message")
```

When `raise` statement is encountered, a *traceback* is printed to the screen, which tells us the sequence of statements that brought us to the `raise` statement, along with function names and line numbers. Finally, the name of the `error_class` and the `error_message` are printed to the screen. The traceback helps find the source of the error, while the error class and message tell us what the error is. The error message is of `string` type—a type for things best represented with words and letters. We will discuss strings in detail later. For now, be aware that a string is a concatentation of characters and whitespace enclosed in either single quotes `''` or double quotes `""`.

```
25      if n < 0:  # if statement
26          raise ValueError("n! is undefined for n less than zero")  # raise statement
```

```
1   # some_code.py
2   from factorial import factorial
3
4   factorial(-1)
```

```
farrelljd@sommerfugl:~/2019/programming_course$ python some_code.py
Traceback (most recent call last):
  File "some_code.py", line 4, in <module>
    factorial(-1)
  File "/home/farrelljd/2019/programming_course/factorial.py", line 29, in factorial
    raise ValueError("n! is undefined for n less than zero")  # raise statement
ValueError: n! is undefined for n less than zero
farrelljd@sommerfugl:~/2019/programming_course$
```

From the traceback, we find that the problem arises from supplying a negative value to `factorial` in line 4 of *some_code.py*.

From the list of built-in exceptions, we find that a `ValueError` is raised when

> "... a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`."

which seems appropriate to our case. If no built-in exception adequately describes the situation, it is possible to define a new one.

```
27      elif not isinstance(n, int):
28          raise TypeError("n is not an integer")  # raise statement
```

A `TypeError` is raised when

> "... an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
>
> This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, NotImplementedError is the proper exception to raise.
>
> Passing arguments of the wrong type (e.g. passing a list when an int is expected) should result in a TypeError, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a ValueError."

which, again, seems to be the right choice.

And that's it for the `factorial` function!

Question 2 The Fibonacci Sequence: part 2

Write a documentation string for your Fibonacci sequence implementation.

I include a code skeleton below, with a sample docstring. The docstring is not complete!

```
def fibonacci(n: int):
    """

    calculates the n^th value of the Fibonacci sequence, F_n

    >>> fibonacci(0)
    0

    >>> fibonacci(1)
    1

    >>> fibonacci(100)
    354224848179261915075

    :param n: element of the Fibonacci sequence to be calculated
    :type n: int

    :return: F_n
    """

    return
```

1.3 Basics II

By the end of this section, you should be able to understand and write python code that looks like the following:

ljpotential.py:

```
def lennard_jones_potential(r: float, epsilon: float = 1.0, sigma: float = 1.0):
    """

    Calculates the Lennard-Jones potential for particles with diameter sigma
    at a separation r with a well-depth epsilon,

        V_{LJ}(r) = 4 epsilon ( (sigma/r)^{12 - (sigma/r)^6 ) .

    >>> lennard_jones_potential(1.0, 1.0, 1.0)
    0.0

    >>> lennard_jones_potential(2**(1/6), 1.0, 1.0)
    -1.0

    >>> lennard_jones_potential(0.0, 1.0, 1.0)
    Traceback (most recent call last):
    ZeroDivisionError: float division by zero

    >>> lennard_jones_potential(-1.0, 1.0, 1.0)
    Traceback (most recent call last):
    ValueError: distance between particles is negative

    >>> lennard_jones_potential(1.0, -1.0, 1.0)
    Traceback (most recent call last):
    ValueError: particle diameter is not strictly positive

    """

    if r < 0.0:
        raise ValueError("distance between particles is negative")
    elif sigma <= 0.0:
        raise ValueError("particle diameter is not strictly positive")

    r6 = (sigma / r) ** 6

    return 4 * epsilon * r6 * (r6 - 1)
```

pairpotential.py:

```
from itertools import combinations
from math import sqrt
from typing import Callable


def pair_potential(xs: [[float]], potential: Callable[[float, ...], float],
                   potential_args: tuple = ()) -> float:

    """

    Calculates the potential energy of configuration of particles.

    >>> from ljpotential import lennard_jones_potential as lj

```

```
    >>> pair_potential(x=[[0.0,0.0,0.0]], potential=lj)
    0.0

    >>> pair_potential(x=[[0.0,0.0,0.0],[0.0,0.0,1.0]], potential=lj)
    0.0

    >>> pair_potential(x=[[0.0,0.0],[0.0,1.0],[0.0,2.0]],
    ... potential=lj,
    ... potential_args=(1.0, 1.0)) # 2D configuration
    -0.0615234375

    :param x: positions of the particles
    :param potential: computes the energy of one pair; must be of the form f(x, *args)
    :param potential_args: arguments to pass to the function

    :return: energy of the configuration
    :rtype: float
    """

    energy = 0.0

    for x1, x2 in combinations(x, 2):  # for statement
        r_squared = 0.0
        for c1, c2 in zip(x1, x2):  # for statement
            r_squared += (c1 - c2) * (c1 - c2)
        r = sqrt(r_squared)  # sqrt imported from math module
        energy += potential(r, *potential_args)

    return energy
```

ljpotential.py contains a function that evaluates the Lennard-Jones potential between two particles at some distance `r`. The Lennard-Jones potential is given by the expression:

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

where σ is the diameter of the particles and ϵ is the depth of the potential well. It finds use as a simple model for pair interactions in noble gases and simple fluids, among other applications. We are already able to fully understand the contents of *ljpotential.py* with the Python from §1.2.

pairpotential.py contains a function that calculates the potential energy of a collection of particles that interact with potential `potential`. The energy of a collection of particles in which only pair-wise interactions occur is given by

$$V_{\text{total}} = \sum_{i \neq j} V(r_{ij})$$

where V is a pair-wise potential energy function, i and j are indices of particles, and r_{ij} is the distance between the particle centres of mass. The Python function `pair_potential` is designed to be agnostic of the specific nature of the interaction, and so can be used in many situations.

The idea of separating out functionality in this way, making code *modular*, is very important regardless of the language used, and can speed up writing, reading, changing, and debugging code dramatically.

1.3.1 Lists

The documentation string of `pair_potential` identifies the variable `x`, the positions of the particles, as a *list of lists*. What is a list of lists? A list whose elements are lists, of course.

In many situations, we want to represent collections of objects in our code. Two examples are the coordinates of a physical system, and a database of customers and their details. We are interested not only in representing the basic elements themselves (the x-coordinate of a particle, which might be represented with a float, or the name of an individual customer, which might be represented with a string), but also

the relationships between them (which x-coordinate is paired with which y-coordinate, which name goes with which phone number). In Python, the job of representing these relationships is done by *collections*, or *container datatypes*.

Different containers are optimised for different tasks. We will not go into any low-level detail about how containers work; we will just cover the basics of how the built-in containers are applied. The curious can read more about other Python containers here.

The aforementioned *list* is a built-in container datatype. A list may contain any number of elements, and has a well-defined order. Let us see some lists in action.

```
>>> []  # the empty list
[]
>>> my_list = [5]
>>> my_list
[5]
>>> my_list.append(8)
>>> my_list
[5, 8]
>>> my_list + my_list  # sensible arithmetic operations are defined
[5, 8, 5, 8]
>>> my_list * 3  # such as multiplication by integers
[5, 8, 5, 8, 5, 8]
>>> [5] + [[5]] + [[[5]]]  # lists can contain other lists, and elements may be of any type
[5, [5], [[5]]]
```

Individual elements of lists may be accessed according to their *index* (their position in the list) *via* the indexing operator, `[]`,

```
>>> my_list = [[1, 2], [3, 4], [5, 6]]
>>> my_list[0]
[1, 2]
>>> my_list[1][1]
4
>>> my_list[0::2]
[[1, 2], [5, 6]]
>>> my_list[::-1]
[[5, 6], [3, 4], [1, 2]]
>>> my_list[0] = 12  # assigning to a element changes the list
>>> my_list
[12, [3, 4], [5, 6]]
```

The expression `my_list[a:b:c]` evaluates to a list containing every c^{th} element of `my_list` starting with the a^{th} element proceeding up to, but not including, the b^{th} element. If c is negative, the order is reversed. If a or b is negative, then the index is with respect to the last element, working backwards, *i.e.*, `l[-1]` evaluates to the last element of `l`, `l[-2]` the second to last, etc.

```
>>> numbers = list(range(12))
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> numbers[1:9:2]
[1, 3, 5, 7]
```

The length of a list (as well as any other object with a `__len__` method—a *sized* object) can be interrogated with the `len` function,

```
>>> l = [0, 1, 2]
>>> len(l)
3
>>> l.append(3)
>>> len(l)
4
```

1.3.2 The `for` statement (definite iteration)

Previously we saw the `while` statement, which executes a block of code until some condition is met. The `for` statement is used when we wish the execute a block of code a *fixed number* of times, for example, once for every atom in a molecule. The `for` statement has the form

```
for target_list in expression_list:
    suite
else:
    suite_b
other code
```

Taken from the Python reference,

> The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The `suite` is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the `suite` in the `else` clause, if present, is executed, and the loop terminates.

Consider the following simple example of taking the arithmetic mean of the elements of a list,

```
1 l = [0, 1, 2, 3, 4]
2 average = 0
3 for x in l:
4     average += x
5 else:
6     average /= len(l)
```

When execution reaches line 3, an iterable is created from the list `l`. The first value in `l`, `0`, is assigned to the name `x`. `x` is then added to average on line 4, which now equals `0`. We have reached the end of the suite, so we return to line 3, where the next value in l, `1`, is assigned to the name `x`. The suite is executed a total of 5 times, once for every element of `l`. Finally, the suite associated with the `else` branch is executed. The `len` function on line 6 takes an iterable as its argument, and returns the number of elements in the iterable (here, 5).

Info: `average += x` and `average /= len(l)` are *augmented assignment statements*. They can be written as the ordinary assigment statements `average = average + x` and `average = average / len(l)`, and achieve the same effect, but are somewhat faster because:

a. they evaluate the target one fewer time (here, the target is `average`);

b. where possible, they directly modify the object to which the target was originally assigned, rather than creating a new object and assigning it to the target.

Using the `len` function, the above `for` loop could be re-written as a `while` loop,

```
l = [0, 1, 2, 3, 4]
average = 0
n = len(n)
i = 0
while i < n:
    average += l[i]
    i += 1
average /= n
```

Of course, the whole exercise could be avoided by using the built-in function `sum`.

1.3.3 Functions of lists

`zip`

The `zip` function allows us to iterate over the elements of two or more lists, pairing elements with the same index.

```
>>> for a, b, c in zip([0, 1, 2], [3, 4, 5], [6, 7, 8]):
...     print(a, b, c)
...
0 3 6
1 4 7
2 5 8
```

On line 36 of *pairpotential.py*, `zip` is used to iterate over the elements of `x1` and `x2`, the coordinates of a pair of distinct particles,

```
36      for x1, x2 in combinations(x, 2):  # for statement
37          r_squared = 0.0
```

Notice that the dimension of the system is not explicitly given; the lists `x1` and `x2` could contain 2, 3, or more float coordinates. This code is designed to work for a system of any dimension. In the three-dimensional case, the suite of the `for` loop starting at line 36 is executed three times, one for each pair of x-, y-, and z-coordinates.

Info: Zippers beware! The iterable returned from by `zip` will yield as many items as there are in the shortest argument passed to it,

```
>>> for a, b, c in zip([0, 1, 2], [3, 4], [6]):
...     print(a, b, c)
...
0 3 6
```

In the above example, the shortest argument, `[6]`, has length one, so the iterable produced yields only one item.

`itertools.combinations`

The `combinations` function in the `itertools` module allows us to iterate over non-identical n-tuples of elements in the same list. `combinations` takes two arguments: a list (or any other *iterable*), and an integer that specifies whether pairs, triples, ..., n-tuples of elements should be returned.

```
>>> for a,b in combinations([0,1,2], 2):
...     print(a,b)
...
0 1
0 2
1 2
>>> for a,b,c in combinations([0,1,2,3], 3):
...     print(a,b,c)
...
0 1 2
0 1 3
0 2 3
1 2 3
```

If tuples including repeated elements are required, the `product` function, also in the `itertools` module, can be used. `product(iterable, repeat=n)` effectively implements an n-level nested loop; *i.e.*,

```
for x, y, z in product(l, repeat=3):
    print(x, y, z)
```

has the same effect as

```
for x in l:
    for y in l:
        for z in l:
            print(x, y, z)
```

The approaches are equally computationally efficient, but the first example is much easier to read. On line 34 of *pairpotential.py*, `combinations` is used to iterate over every pair of non-identical particles,

```
34      energy = 0.0
35
36      for x1, x2 in combinations(x, 2):  # for statement
37          r_squared = 0.0
38          for c1, c2 in zip(x1, x2):  # for statement
39              r_squared += (c1 - c2) * (c1 - c2)
```

If the system contains n particles, the suite of the `for` loop beginning at line 34 is executed $\binom{n}{2}$ times.

1.3.4 Functions and their arguments

What now remains is to explain the function call to `potential` on line 39. From the perspective of the `pair_potential` function, it doesn't matter how many arguments the *callable* `potential` might take, or what their names may be—all that matters is that the arguments are passed to `potential`, and in the correct order (however, the type hint indicates that the function should accept at least one float argument, and return a float).

The arguments are passed to `pair_potential` as a tuple, which is *unpacked* in the function call to `potential` with the `*` operator. The call

```
pair_potential(x=[[0.0,0.0],[0.0,1.0],[0.0,2.0]], potential=lj, potential_args=(1.0, 1.0))
```

results in the call

```
energy += potential(r, 1.0, 1.0)
```

This syntax allows the function `pair_potential` the flexibility to work with more than one kind of *function signature* (here, potential functions with different numbers of arguments).

Info: Like a `list`, a `tuple` is a container type. Unlike lists, tuples are *immutable*—once a tuple object has been created, it cannot be modified. We can use `len` to find the length of a tuple, and `[]` to access the elements of a tuple, but, since they are immutable, assigning to an element raises an exception, and the `append` method is not defined.

```
>>> ()  # empty tuple
()
>>> (1,)  # single-element tuple---notice the terminal comma, ","!
(1,)
>>> (1)  # this is an integer!
1
>>> (1, 2)  # two-element tuple---no terminal comma necessary
(1, 2)
>>> tup = (1, 2, 3)
>>> tup
(1, 2, 3)
>>> len(tup)  # length
3
>>> tup[0]  # indexing
1
>>> tup[0::2]  # slicing
(1, 3)
>>> tup[0] = 4  # assigning---not supported
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tup.append(3)  # appending---not supported
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> tup[0]
```

In some situations, a list is the most appropriate container type; in others, a tuple is preferred. For example, when it is necessary to modify the contents of the container, a list is more useful than a tuple. When it is necessary that the elements of the container be read-only, a list *can* be used, but a tuple is more suited to the problem, as an error is raised when writing to an element is attempted.

Question 3 Prime Numbers

Using your new-found knowledge of container datatypes, write a function that returns a list of its argument's prime factors, e.g.,

```
>>> prime_factors(1)
[]
>>> prime_factors(2)
[2]
>>> prime_factors(60)
[2, 3, 5]
```

Include a documentation string with tests and **make sure your code passes its own tests!**

Optional: write a function `is_prime`, that returns `True` if its argument is prime, and `False` otherwise. Make your function as efficient as possible.

1.4 Scientific programming

By the end of this section, you should be able to understand all of the python code in `simulations.potentials` and `simulations.pair_potential`, and have a vague understanding of what code like *timing_pair_potential.py* is supposed to do.

The original `simulations.pair_potential` function calculates the pairwise-additive energy of a system of particles interacting *via* an unspecified `potential` function. This calculation is done *via* a pair of nested loops: the outer loop to iterate over particle pairs; the inner loop to iterate over particle coordinates,

```
6  def pair_potential(xs: [[float]], potential: Callable[[float, ...], float],
7                     potential_args: tuple = ()) -> float:
34     energy = 0.0
35
36     for x1, x2 in combinations(x, 2):  # for statement
37         r_squared = 0.0
38         for c1, c2 in zip(x1, x2):  # for statement
39             r_squared += (c1 - c2) * (c1 - c2)
40         r = sqrt(r_squared)  # sqrt imported from math module
41         energy += potential(r, *potential_args)
42
43     return energy
```

(docstrings omitted for brevity).

The function `pair_potential_half_vectorised` achieves the same effect, but replaces the inner loop with a *vectorised* calculation using the `numpy` package.

```
8  def pair_potential_half_vectorised(xs: ndarray, potential: Callable[[float, ...], float],
9                                     potential_args: tuple = ()) -> float:
36     energy = 0.0
37
38     for x1, x2 in combinations(xs, 2):  # for statement
39         r = np.linalg.norm(x1-x2)
40         energy += potential(r, *potential_args)
41
42     return energy
```

`pair_potential_vectorised` is a similar function with both loops replaced with vectorised code, the result being significantly faster than our original `pair_potential` implementation.

```
7  def pair_potential_vectorised(xs: ndarray, potential: Callable[[ndarray, ...], ndarray],
8                                potential_args: tuple = ()) -> float:
35     nparticles, ndim = xs.shape
36     left_indices, right_indices = np.triu_indices(nparticles, k=1)
37     rij = xs[left_indices] - xs[right_indices]
38     dij = np.linalg.norm(rij, axis=1)
39
40     return potential(dij, *potential_args).sum()
```

To demonstrate this difference, the script *timing_pair_potential.py* contains functions to compare the speed of the three implementations as a function of particle number, and plots the results using the `matplotlib` package.

1.4.1 NumPy

NumPy, short for **Num**erical **Py**thon, is a python package for scientific programming. It provides a framework for efficiently dealing with large amounts of data, and implements many algorithms from linear

algebra, Fourier transforms, and efficient random number generations, to name just three fields. The key component of the NumPy module is the *n-dimensional array*, or *ndarray*.

The ndarray, like the list, is a container type, but with several important differences. Ndarrays can be initialised with the `array` function, which takes a list as its argument,

```
>>> import numpy as np
>>> a = np.array([0, 1, 2])
>>> type(a)
<class 'numpy.ndarray'>
>>> a
array([0, 1, 2])
```

The list can contain any numeric type, and the list elements can be cast to another type with the `dtype` keyword argument,

```
>>> a = np.array([0.0, 1.0, 2.0])
>>> a
array([ 0.,  1.,  2.])
>>> a = np.array([0, 1, 2], dtype=float)
>>> a
array([ 0.,  1.,  2.])
>>> a = np.array([0, 1, 2], dtype=complex)
>>> a
array([ 0.+0.j,  1.+0.j,  2.+0.j])
```

N-dimensional ndarrays can be initialised by passing a list of lists (of lists…etc) as the argument,

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

The dimensions of the ndarray are stored in the `shape` attribute, and the total number of elements can be accessed *via* the `size` attribute. The shape can be changed *via* the `reshape` method,

```
>>> a = np.array([[[0, 1], [2, 3], [4, 5]]])
>>> a
array([[[0, 1],
        [2, 3],
        [4, 5]]])
>>> a.shape
(1, 3, 2)
>>> a.size
6
>>> b = a.reshape(3, 1, 2)
>>> b
array([[[0, 1]],

       [[2, 3]],

       [[4, 5]]])
>>> b.shape
(3, 1, 2)
>>> c = a.reshape(-1)
>>> c
```

```
array([0, 1, 2, 3, 4, 5])
>>> c.shape
```

Using the `len` function on arrays will return the left-most element of the shape tuple,

```
>>> a = np.array([[[0, 1], [2, 3], [4, 5]]])
>>> len(a) == a.shape[0]
True
```

While lists can contain objects of any type, ndarrays are homogeneous. If the elements of the argument list can be cast to the same dtype, an ndarray with that dtype will be created,

```
>>> a = np.array([1, 2, "string"])  # array of strings!
>>> a
array(['1', '2', 'string'],
      dtype='<U21')
```

Numeric-type ndarrays are contiguous, *i.e.*, have a rectangular shape. If a list of lists of different lengths is passed as the argument, an object-type ndarray is created,

```
>>> a = np.array([[[0,0],[1,1]],[[0,1,2,3],[0,1,2,3]]])
>>> a
array([[[0, 0], [1, 1]],
       [[0, 1, 2, 3], [0, 1, 2, 3]]], dtype=object)
>>> a.shape
(2, 2)
```

If the lists have different depths (*e.g.* a list and a list of lists), an exception is raised,

```
>>> a = np.array([1, [2], [[3]]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: setting an array element with a sequence.
```

Ndarrays filled with zeros or ones can by constructed with the `zeros` and `ones` functions, as well as the `zeros_like` and `ones_like` functions. These functions are useful for initialising arrays. The former take the shape of the array as the first argument, and an optional dtype; the latter take another array whose shape is to be copied, along with an optional dtype,

```
>>> a
array([ True,  True,  True,  True,  True], dtype=bool)
>>> b = np.zeros_like(a, dtype=float)
>>> b
array([ 0.,  0.,  0.,  0.,  0.])
```

ndarrays support the same kind of indexing as lists, as well as `fancy indexing`, which allows a list or another ndarray to be passed as the argument to the indexing operator,

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a[0::2]
array([0, 2, 4])
>>> a[[0, 3, 4]]
array([0, 3, 4])
>>> a[np.array([[0, 0], [1, 1], [2, 2], [3, 3]])]
array([[0, 0],
       [1, 1],
       [2, 2],
       [3, 3]])
```

Normal indexing returns a *view* of an array; changes to a view of an array are also applied to the array itself,

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> b = a[0::2]
>>> b
array([0, 2, 4])
>>> b[0] = 6
>>> a
array([6, 1, 2, 3, 4, 5])
```

Fancy indexing creates a *copy* of an array; a new array object,

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> b = a[[0,2,4]]
>>> b[0] = 6
>>> a
array([0, 1, 2, 3, 4, 5])
```

Be careful to take notice whether you are working with views or copies.

Tip If an array view is passed to a function, and the function changes the view, the orignal array will also be changed! This is also true of lists, and a source of consternation for beginners and experts alike. If a function changes its arguments rather than creating a new object, it is said to change the argument *in place*. A pure function does not change its arguments; impure functions are not allowed in programs written in purely functional languages. Python is very liberal in this regard—it is up to you to make sure that the user knows whether a function modifies its arguments in place, by writing clear documentation.

Operations

The arithmetic and comparison operators work on an element-wise basis,

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> a * 5
array([ 0,  5, 10, 15, 20, 25])
>>> b = np.array([5, 4, 3, 2, 1, 0])
>>> a * b
array([0, 4, 6, 6, 4, 0])
>>> a ** b
array([0, 1, 8, 9, 4, 1])
>>> a < b
array([ True,  True,  True, False, False, False], dtype=bool)
```

The boolean array that results from a comparison operation can be used as a filter,

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> mask = a < 4
>>> a[mask]
array([0, 1, 2, 3])
```

Since python3.5, matrix multiplication can be acheived with the `@` operator,

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a @ a
array([[ 7, 10],
       [15, 22]])
>>> b = np.array([0, 1, 2])
>>> b @ b  # dot product
5
```

For arithmetic and comparison operators to succeed, the two arrays must have the same shape; otherwise, a `ValueError` is raised. Dummy dimensions can be added by passing `None` or `np.newaxis` to the indexing operator,

```
>>> a = np.array([0, 1, 2])
>>> b = np.array([[0, 1, 2]])
>>> a @ b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (3,) and (1,3) not aligned: 3 (dim 0) != 1 (dim 0)
>>> a[:, np.newaxis] @ b
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
>>> a[:, np.newaxis] @ a[np.newaxis, :]  # outer product
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
>>> a[np.newaxis, :] @ a[:, np.newaxis]  # inner product
array([[5]])
```

Functions

NumPy provides many useful functions for operating on ndarrays. There are NumPy versions of many built-in functions and functions in built-in modules,

```
>>> import numpy as np
>>> a = np.array([0,1,-2])
>>> np.abs(a)
array([0, 1, 2])
>>> np.sum(a)
3
>>> np.cos(a)
array([ 1.        ,  0.54030231, -0.41614684])
```

as well as other convenient definitions,

```
>>> np.std(a)  # standard deviation of a sample
0.816496580927726
>>> np.linalg.norm(a)  # norm of a vector
2.23606797749979
```

When applying reduction operations (`np.sum` , `np.linalg.norm` , etc), an axis can be specified,

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> np.sum(b, axis=0)
array([12, 15, 18, 21])
>>> np.linalg.norm(b, axis=1)
array([ 3.74165739, 11.22497216, 19.13112647])
```

Attempting to apply a function from the `math` module to an ndarray will usually result in a `TypeError` ,

```
>>> from math import cos
>>> cos(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only size-1 arrays can be converted to Python scalars
```

Vectorisation

The NumPy operations and functions we have seen so far are *vectorised*. Let's say we want to calculate the cosine of a thousand numbers in an ndarray. We could iterate over this array in a `for` loop, calculating the cosine of each number in sequence, and storing it somewhere useful. However, since the cosine of the first number doesn't depend on the cosine of the second, third ... or thousandth number, we could, in principle, calculate them all at the same time! That's what happens when we call the vectorised `np.cos` function. Independent cosines are computed simultaneously, with the number of concurrent calculations depending on the hardware. The result, for medium to large arrays, is a significant reduction in compute time. For small arrays, the difference might be unnoticeable, and for very small arrays, the vectorised calculation may in fact be slower.

Now we return to our `pairpotential*` series, to see how vectorisation can help us to write efficient code.

The original `pair_potential` contains a two-level nested loop,

```
34      energy = 0.0
35
36      for x1, x2 in combinations(x, 2):  # for statement
37          r_squared = 0.0
38          for c1, c2 in zip(x1, x2):  # for statement
39              r_squared += (c1 - c2) * (c1 - c2)
40          r = sqrt(r_squared)  # sqrt imported from math module
41          energy += potential(r, *potential_args)
```

Lines 34–39 accumulate the squared distance between two particles in a loop, considering each dimension sequentially, and find the distance as its square root. The function `pair_potential_half_vectorised` in instead contains a call to `np.linalg.norm`, calculating the interparticle distance directly from the particle coordinate arrays,

```
38      for x1, x2 in combinations(xs, 2):  # for statement
39          r = np.linalg.norm(x1-x2)
40          energy += potential(r, *potential_args)
```

The remaining loop can be removed by noticing the following: we want to express the entire distance calculation in a series of vector operations. We seek an expression of the form,

$$r_{ij} = r_j - r_i$$

where the rows of r_i and r_j put together each constitute a unique interparticle vector.

We achieve this through fancy indexing. The indices we want are the row and column indices of the upper triangle of an $N \times N$ matrix, without the diagonal indices (self interactions), where N is the number of particles (the first element of `x.shape`).

The functions `pair_potential_vectorised` obtains these indices using the NumPy function `triu_indices`, which returns two arrays containing the relevant row and column indices,

```
35      nparticles, ndim = xs.shape
36      left_indices, right_indices = np.triu_indices(nparticles, k=1)
37      rij = xs[left_indices] - xs[right_indices]
38      dij = np.linalg.norm(rij, axis=1)
```

We then compute the distances with `np.linalg.norm`, reducing over the last axis.

This description may be somewhat opaque, so let's look at an example. If we have three particles, we need to calculate three unique distances, between particles one and two, one and three, and two and three. So, in r_i we want the rows to be the coordinates of particles (1, 1, 2). Likewise, in r_i we want the rows to be the coordinates of particles (2, 3, 3). The call `triu_indices(3, k=1)` provides exactly that,

```
import numpy as np
>>> coordinates = np.array(["one", "two", "three"])
>>> left_indices, right_indices = np.triu_indices(3, k=1)
>>> left_indices
```

```
array([0, 0, 1])
>>> right_indices
array([1, 2, 2])
>>> coordinates[left_indices]
array(['one', 'one', 'two'],
      dtype='<U5')
>>> coordinates[right_indices]
array(['two', 'three', 'three'],
      dtype='<U5')
```

Truth

Do we need to change `lj_potential` to take advantage of vectorisation? A little.

Since there are no loops in `lj_potential`, and arithmetic and comparison operators are already vectorised, we don't need to worry about the implementation of the energy calculation.

There are two small changes that need to be made. We need to modify our doctests to use arrays,

```
    >>> from numpy import array
    >>> lj_potential_vectorised(array([1.0, 2**(1/6), 2.0]), epsilon=1.0, sigma=1.0)  #doctest: +ELLIPSIS
    array([ 0.        , -1.        , -0.0615...])
```

and to make some changes to our exceptions.

Truth testing is ambiguous for arrays with more than one element. That is to say, an ndarray with more than one element is neither true nor false. As such, the statement `if r < 0.0:` in the original `lj_potential` will raise a `ValueError`. To test if any, or all elements of an array are "true", we can use the `any` and `all` functions. `any(a)` returns `True` if at least one element of `a` is a true value, and `False` otherwise; `all(a)` returns `True` if all elements of `a` are true values, and `False` otherwise.

In our case, we want to check whether *any* element of `r` is less than zero, and whether *any* element of `sigma` is less than or equal to zero, so we use the *np.any* function,

```
    if not isinstance(rs, ndarray):
        raise TypeError(f'rs should be ndarray, not {type(rs).__name__}')
    if np.any(rs <= 0):
        raise ValueError(f'all r must be positive, but min(r) = {rs.min()}')
    if epsilon <= 0 or sigma <= 0:
        raise ValueError(f'both epsilon and sigma must be positive, not ({epsilon}, {sigma})')
```

In addition to `np.any` and `np.all`, the `np.where` function can be used to effect element-wise `if...else`,

```
>>> a = np.array([0, 1, 2, 3, 4, 5])
>>> np.where(a > 3, 3, a)
array([ 0,  1,  2,  3,  3,  3])
```

Our new vectorised version of `lj_potential` is then,

```
import numpy as np
from numpy import ndarray


def lj_potential_vectorised(rs: ndarray, epsilon: float = 1.0, sigma: float = 1.0) -> ndarray:
    """
    compute the Lennard Jones potential at particle separation r,

    V_LJ = 4 epsilon ( (sigma/r)^12 - (sigma/r)^6 )

    >>> from numpy import array
```

```
    >>> lj_potential_vectorised(array([1.0, 2**(1/6), 2.0]), epsilon=1.0, sigma=1.0)  #doctest: +ELLIPSIS
    array([ 0.        , -1.        , -0.0615...])

    >>> lj_potential_vectorised(-1)
    Traceback (most recent call last):
    TypeError: rs should be ndarray, not int

    >>> lj_potential_vectorised(array([1.0, 2**(1/6), -1.0]))
    Traceback (most recent call last):
    ValueError: all r must be positive, but min(r) = -1.0

    >>> lj_potential_vectorised(array([1.0, 2**(1/6), 2.0]), epsilon=-1.0, sigma=1.0)
    Traceback (most recent call last):
    ValueError: both epsilon and sigma must be positive, not (-1.0, 1.0)
    """

    if not isinstance(rs, ndarray):
        raise TypeError(f'rs should be ndarray, not {type(rs).__name__}')
    if np.any(rs <= 0):
        raise ValueError(f'all r must be positive, but min(r) = {rs.min()}')
    if epsilon <= 0 or sigma <= 0:
        raise ValueError(f'both epsilon and sigma must be positive, not ({epsilon}, {sigma})')

    r6 = (sigma / rs) ** 2
    r6 *= r6 * r6
    return 4 * epsilon * r6 * (r6 - 1)
```

1.4.2 Matplotlib

Matplotlib is a python package that provides functions to draw figures. You can get started with Matplotlib very quickly,

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> a = np.linspace(0,1,100)
>>> plt.plot(a, a*a)
>>> plt.show()
```

Here, the function `plt.plot` is given two positional arguments, which are arrays of equal size. This function creates the plot object, but `plt.show` must be called to display the plot. If you type the above commands into the console, you should get a plot of $y = x^2$ for $x \in [0, 1]$.

To have more control over your figures, it is best to instantiate a `figure` object and accompanying `axis` objects, as in *timing_pair_potential.py*,

```
    fig, ax = plt.subplots(1, 1, dpi=160, constrained_layout=True, figsize=plt.figaspect(1 / 2))
    ax2 = plt.twinx(ax)
```

where we have created a figure two twinned axes.

Axis labels and other properties of axes can be set *via* a variety of `.set_*` methods of axes objects, and lines can be labelled by passing a `label` keyword argument to any of the many plotting functions (if you add labels, don't forget to call `ax.legend`. We move all of these details into a separate function, `modify_plot`,

```
def modify_plot(fig, ax, ax2, nparticles):
    ax.set_xlabel('number of particles')
    ax.set_ylabel('time / seconds')
    ax.set_xscale('log')
```

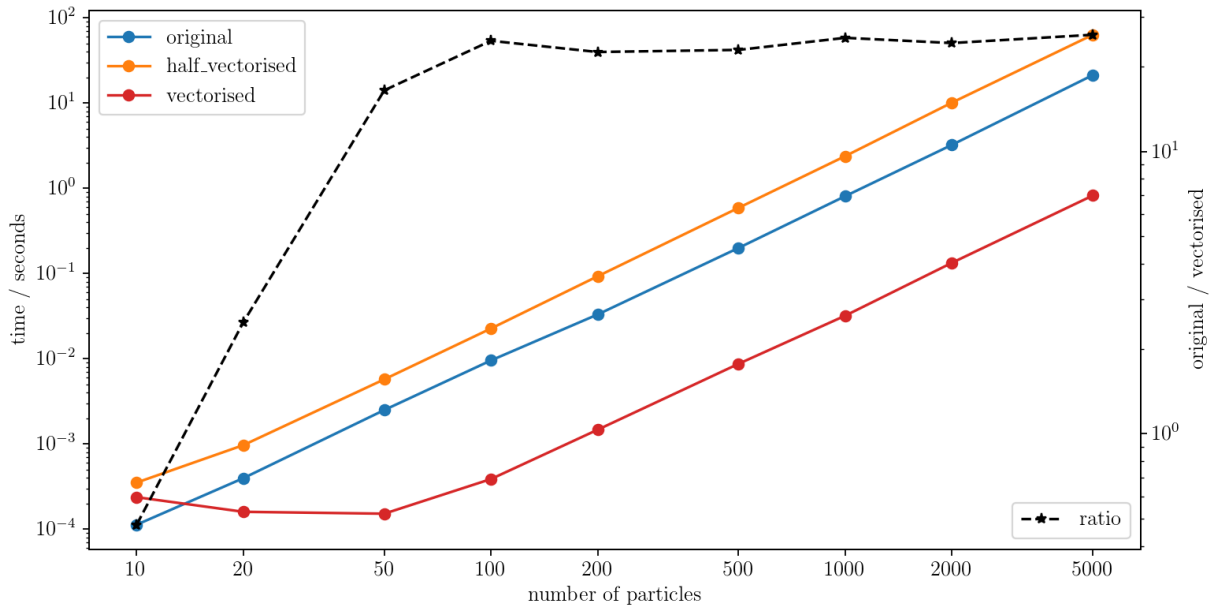

Figure 1.1: Upper panel: A comparison of the execution times of three implementations of the pair potential function mentioned in the text. Lower panel: relative execution time of the two loop implementation compared to the vectorised implementation.

```
36      ax.set_yscale('log')
37      ax.set_xticks(nparticles)
38      ax.set_xticks([], minor=True)
39      ax.set_xticklabels(nparticles)
40      ax.legend()
41
42      ax2.set_ylabel('original / vectorised')
43      ax2.set_yscale('log')
44      ax2.legend(loc=4)
45      return ax
```

Different line styles can be included in call to the plot function (see the Matplotlib documentation for details).

To save a figure rather than plot it to the screen, the `plt.save_fig` function is used, with the file name as its first argument.

Running *timing_pair_potential.py* will compare the execution speed of our three pair potential functions, `pair_potential` , `pair_potential_halfvectorised` , and `pair_potential_vectorised` for different numbers of particles, and save a figure to *timing_pair_potential.png*.

Give it a try! My result is shown figure 1.1, and indicates that, on my desktop, the vectorised code we saw in this section executes about 25 times faster than that from the last section for large particle number.

Info: The `__main__` block of *timing_pair_potential.py* is executed when *timing_pair_potential.py* is the main program; if the main program imports something from *timing_pair_potential.py*, the `__main__` block will be ignored.

Question 4 Molecular Dynamics Prep.

1. **Forces.** Write a function that calculates the per-particle forces in a system of Lennard-Jones particles. First write a naive loop to make sure you get the right result, then implement a vectorised version.

Chapter 2

Numbers and Precision

2.1 Decimal Numbers

There are many numbers—uncountably many, in fact. If a language were to assign to each number a unique word, that language would similarly have infinitely many words. In China, a senior high-school student is expected to learn ≈ 6600 characters, yet most can discuss numbers beyond 10000.

Most advanced civilisations have relied upon number systems that represent numbers as some expansion of powers of some base. For example, in Ancient Egypt, powers of ten from one to one million each had their own symbol (hieroglyph), with other numbers expressed by writing the appropriate number of copies of those basic symbols (table 2.1). Such a system has the number ten as its base, and is referred to as a decimal number system.

In the prevailing positional decimal number system, integers are expressed as decimal numerals, ordered lists of numerals 0–9, encoding a decimal expansion,

$$a_n a_{n-1} \dots a_1 a_0 = \sum_{i=0}^{n} a_i \times 10^i. \tag{2.1}$$

Real numbers may be expressed as two such numerals interposed by a decimal point, .,

$$a_n a_{n-1} \dots a_1 a_0 . b_1 b_2 \dots b_{m-1} b_m = \sum_{i=0}^{n} a_i \times 10^i + \sum_{j=1}^{m} b_j \times 10^{-j} \tag{2.2}$$

Typically, n, m are chosen such that $a_n \neq 0$, $b_m \neq 0$. The decimal expansion of irrational numbers, such as π, is infinite. Such numbers are usually quoted to an appropriate number of significant digits, *e.g.*, $\pi = 3.1415$ (5 s.f.). Additionally, some rational numbers, such as $1/7$, have infinitely repeating decimal expansions; to handle such cases, a variety of notations are used, such as writing a line over the repeating sequence,

$$\frac{1}{70} = 0.0\overline{142857}. \tag{2.3}$$

2.2 Binary Numbers

The popularity of the base-ten number system may have something to do with the fact that humans each possess ten fingers (digits). Bases eight, twelve, twenty, and sixty have also seen widespread use. Modern

Arabic Numeral	1	10	100	1000	10000	100000	1000000
Egyptian Hieroglyph							
Description	single stroke	cattle hobble	coil of rope	lotus	bent finger	tadpole	Heh

Table 2.1: The Ancient Egyptian decimal system. The number 2021 is written as  .

computing is almost entirely based on the base-two, or binary number system, owing to the ease with which this system can be implemented in circuits using logic gates.

In a positional binary system, each digit represents a different power of two. For instance,

$$1010_2 = 2^3 + 2^1 = 10, \tag{2.4}$$

where the subscript of the left-hand-side indicates the base. Binary fractions are similar have a similar interpretation to their decimal counterparts,

$$10.101_2 = 2^1 + 2^{-1} + 2^{-3} = 2 + \frac{1}{2} + \frac{1}{8} = 2.625 \tag{2.5}$$

In Python, a binary integer literal is the flag `0b` followed by a sequence of zeros and ones. The binary representation of a decimal number can be obtained (as a string) *via* the `bin` function,

```
>>> 0b1010
10
>>> bin(10)
'0b1010'
```

Info: The equivalent flags and functions for octal (base 8) and hexadecimal (base 16) are `0o, oct` and `0x, hex`.

Any finite binary fraction can be expressed as a finite decimal fraction,

$$\sum_i b_i \times 2^{-i} = \sum_i b_i \times \frac{1}{2^{-i}} \frac{10^i}{10^i} = \sum_i 5^i b_i \times \frac{1}{10^{-i}}, \tag{2.6}$$

but the converse is not true—in base two, the decimal number 0.1 is an infinitely repeating fraction,

$$0.1 = 0.0\overline{0011}_2 \tag{2.7}$$

Although 0.1 and numbers like it cannot be represented exactly as binary fractions, it is still possible to represent them exactly as tuples of integers (a numerator and a denominator).

2.3 Integers

The values representable by an integer type are determined by (1) the size of that type, *i.e.*, the amount of memory allocated to it, and (2) whether or not the type is signed. For instance, a 32-bit unsigned integer i can take values between $0 \leq i < 2^{32}$; each of the 32 bits represents a term in the binary expansion of the number. A 32-bit signed integer j can take values between $-2^{31} \leq j < 2^{31}$; here, 31 bits each represent a term in the expansion, and one bit encodes the sign.

In Python3, the default integer type is unbounded, *i.e.*, there is no limit on the numerical size of the number (hardware limitations still apply). However, numpy integers are bounded. If you try to exceed the bounds of a bounded integer type, you will discover that integer arithmetic is actually implemented as modular arithmetic,

```
>>> from numpy import full, int32, int64
>>> arr = full(1, 2**31-1, dtype=int32)
>>> arr
array([2147483647], dtype=int32)
>>> arr+1
array([-2147483648], dtype=int32)
```

Exceeding the bound of a type is handled be wrapping around to the other bounding value. This situation is known as *integer overflow*.

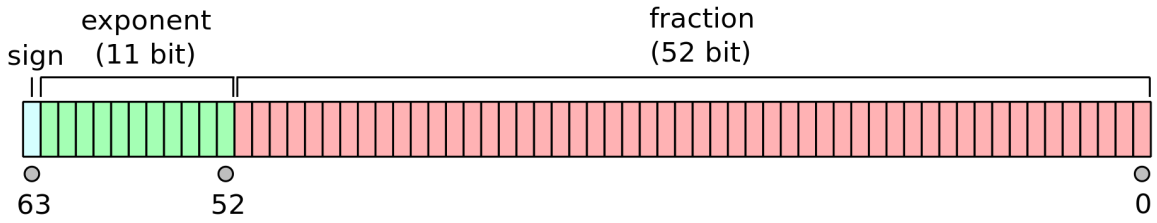

Figure 2.1: The anatomy of a IEEE-754 double-precision float: 1 bit is reserved for the sign, 11 bits for the exponent, and 52 bits for the fraction. If the exponent is non-zero, the number is normal.
Figure taken from here.

2.4 Floating Point Numbers

Numbers other than the integers can be represented using floating-point types. It is often convenient to write such numbers in scientific notation, *i.e.*, a product of a sign, a fractional part (the mantissa), and a base (radix) taken to the power of some exponent,

$$(\text{sign})\,\text{fraction} \times \text{base}^{\text{exponent}} \tag{2.8}$$

The number $-1/8$ is thus represented as -1.25×10^{-1} in decimal and $-1_2 \times 2^{-3}$ in binary.

The most widely-used are the single- and double-precision floating-point numbers (singles and doubles) specified in the IEEE-754 standard. Singles require 32 bits per number: the largest bit encodes the sign, the next 8 bits encode the exponent, and the last 23 bits encode the fraction. Doubles require 64 bits: 1 for the sign, 11 for the exponent, and 52 for the fraction (figure 2.1). Because they involve manipulating less data, operations between singles are faster than between doubles (at least twice as fast, sometimes up to 32 times faster), but computations with doubles are more precise (they can be specified to a larger number of significant figures); in computational physics, the need for higher precision usually wins out, and so doubles are preferred over singles.

2.4.1 Fractional Part

The fractional part, b, actually encodes a 53-bit number, with a largest bit equal to 1,

$$(1.b_{52}b_{51}\ldots b_2b_1)_2 \tag{2.9}$$

Doubles therefore have 53-bit precision in binary, corresponding to 15–17 significant decimal digits. The maximum relative rounding error when rounding a number to the nearest representable one (known as the machine epsilon) is therefore $\epsilon = 2^{`53} \approx 1.11 \times 10^{-16}$.

2.4.2 Exponent

The range of the exponent string, e, is 0–2047. There are several cases to consider.

- $\mathbf{0 < e < 2047}$ the exponent is given by $e - 1023$, *i.e.*, allowable exponents are integers on the range -1022–1023. In this case, the number encoded is *normal*: it has full precision. The rule for decoding the number is the expression,
$$(-1)^s \times 2^{e-1023} \times (1.b)_2 \tag{2.10}$$
where s is the sign bit. The range of the positive normal numbers is thus 2^{-1022} to 2^{1023}, approximately $2.2250738585072014 \times 10^{-308}$ to $1.7976931348623157 \times 10^{308}$ in decimal.

- $\mathbf{e = 0; b \neq 0}$ these are the **denormal** numbers. Denormal numbers fill the underflow gap around zero, ensuring that the difference between two distinct normal numbers is always non-zero. Denormal numbers are defined to less than 53 bits of precision. The expression to decode the string becomes,
$$(-1)^s \times 2^{-1022} \times (0.b)_2 \tag{2.11}$$

- $\mathbf{e = 0; b = 0}$ the number evaluates to ± 0

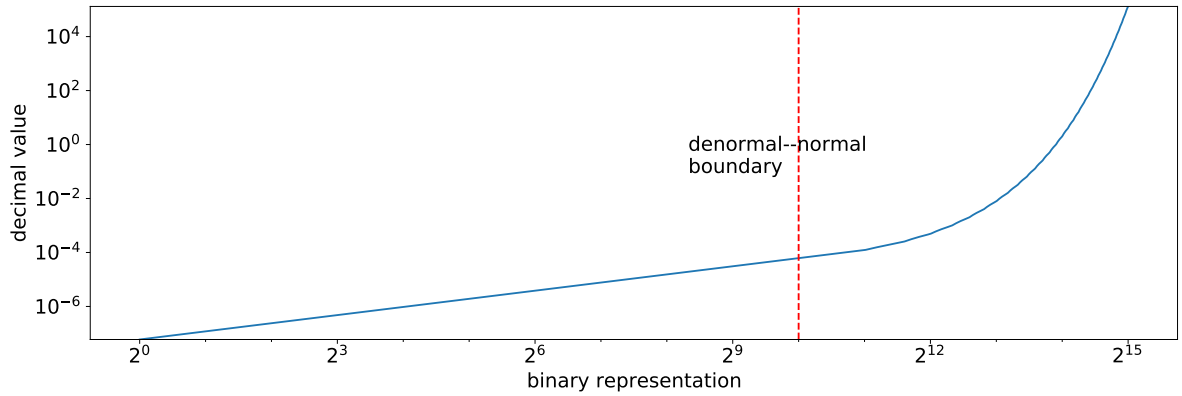

Figure 2.2: Distribution of positive values of 16-bit floats. Subnormal values are linearly spaced, whereas normal values are logarithmically spaced.

- $\mathbf{e} = \mathbf{2047}; \mathbf{b} = \mathbf{0}$ the number evaluates to $\pm\infty$
- $\mathbf{e} = \mathbf{2047}; \mathbf{b} \neq \mathbf{0}$ the number evaluates to NaN (not a number)

Per these definitions, normal floating-point numbers provide a logarithmically-spaced coverage of the number line, whereas the denormal numbers are linearly-spaced. Figure 2.2 illustrates this disparity for positive half-precision floats (16-bits, 1 for the sign, 5 for the exponent, and 10 for the fraction).

2.5 Representation Error and Truncation

As we mentioned in section 2.2, the decimal fraction 0.1 has no finite binary expansion, and as such cannot be represented exactly, and must be truncated in order to be stored; this is a source of representation error. Instead, an input value like 0.1 is converted to the closest binary fraction that fits into the target type. If the target is a double, that means finding the 53-bit integer J and exponent N such that $1/10 \approx J/2^N$, or, rearranging, $J \approx 2^N/10$. Noticing that $8 < 10 < 16$,

$$2^{52} = 2^{N-4} < 2^N/10 < 2^{N-3} = 2^{53} \tag{2.12}$$

so $N = 56$ is the only value for N leaving J with exactly 53 bits. Rounding $2^{56}/10$ to the nearest integer gives

$$J = 7205759403792794 \tag{2.13}$$

As such, best representation of 0.1 in double precision is given by the fraction

$$\frac{7205759403792794}{2^{56}}. \tag{2.14}$$

In Python, we can find the fractional representation of any float using the `to_integer_ratio` method of the `float` type, and the exact decimal value of the stored binary fraction using the `from_float` class method of the `Decimal` class,

```
>>> from decimal import Decimal
>>> x = 0.1
>>> x.as_integer_ratio()
(3602879701896397, 36028797018963968)
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Finally, encoded as a double precision value,

$$0.1 \mapsto 0\,01111111011\,1001100110011001100110011001100110011001100110011010_2. \tag{2.15}$$

2.5.1 $0.1 + 0.1 + 0.1 = 0.3$?

A stark example of representation error is found in the simple statement:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Interrogating the exact decimal expansions of the LHS and RHS, we find that there is a more accurate representation of the number 0.3 than is obtained by the sum $0.1 + 0.1 + 0.1$,

```
>>> Decimal.from_float(0.1+0.1+0.1)
Decimal('0.3000000000000000444089209850062616169452667236328125')
>>> Decimal.from_float(0.3)
Decimal('0.2999999999999999888977697537484345957636833190917968875')
```

Errors like this accumulate over the course of a calculation. For example, to find the mean of the values in a list, you might iterate over the values in the list, adding each to a running total, before finally dividing by the number of values,

```
1  def naive_mean(iterable):
2      total = 0
3      for value in iterable:
4          total += value
5      return total / len(iterable)
```

Applying this to a list of just over a million copies of the float 0.1, we obtain a less than satisfactory result,

```
>>> from sums import naive_mean
>>> values = 2 ** 20 * [0.1]
>>> m = naive_mean(values)
>>> m
0.10000000000154079
>>> abs(0.1-m)
1.5407813913626e-12
```

The error in the mean is four orders of magnitude greater than the machine epsilon. This occurs because when two floats of differing orders of magnitude are added together, some of the bits of the smaller number have to be discarded in order for the sum to fit into the data type. Since 0.1 has only an approximate value as a binary fraction, half of the smallest significant bits are non-zero, and the smallest of them will be lost when adding to the running total.

The problem of lost significance can be changing the order of summation. For example, instead of keeping one running total, we could add together pairs of adjacent elements in the list, producing a new list of half the size. We repeat the procedure on the new list until the new list has only one element, and then divide this element by the length of the original list.

```
8  def clever_sum(iterable):
9      if len(iterable) == 1:
10         return iterable[0]
11     elif (len(iterable) % 2) == 1:
12         return iterable[0] + clever_sum(iterable[1:])
13     else:
14         return clever_sum([a + b for (a, b) in zip(iterable[0::2], iterable[1::2])])
15
16
17 def clever_mean(iterable):
18     return clever_sum(iterable)/len(iterable)
```

This recursive algorithm produces much better results than the naive implementation,

```
>>> from sums import clever_mean
>>> m = clever_mean(values)
>>> m
0.1
>>> abs(0.1-m)
0.0
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

but probably has terrible performance characteristics. The `numpy.mean` method is a better alternative to the builtin sum function, producing an error close to the machine epsilon,

```
>>> import numpy as np
>>> m = np.mean(values)
>>> abs(0.1-m)
2.220446049250313e-16
```

2.6 Catastrophic Cancellation

Catastrophic cancellation is the phenomenon that subtracting good approximations to two nearby numbers may yield a very bad approximation to the difference of the original numbers. Catastrophic cancellation can occur whenever approximations are made, such as with the approximate representation of the number line provided by floating point numbers. Take as a simple example the discriminant of the quadratic equation,

$$ax^2 + bx + c = 0 \to x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.16}$$

If $|4ac/b^2| < \epsilon \approx 10^{-16}$, the one of the roots $x_{1,2}$ will suffer from catastrophic cancellation as its numerator will evaluate to 0. Any further computation that relies on the reciprocal the root will cause an exception, but worse would be for the cancellation to go unnoticed and propagate unhindered.

As we re-wrote the algorithm to compute the sum in the previous section, the solution here is to rewrite one of the roots as a function of the other. Noticing that the product of the roots is always $x_1 x_2 = c/a$, we can evaluate the non-pathological root, say, x_1, and then compute the other as $x_2 = c/ax_1$. The roots are then $x_1 = -b/a, x_2 = -c/b$.

Question 5 Evaluating Functions

Rewrite the following expressions so that they can be safely evaluated in the given limit.

For small h,

$$\sin(x+h) - \sin(x)\,; \tag{2.17}$$

for small x,

$$\frac{1-\cos x}{\sin x}; \tag{2.18}$$

for large N,

$$\int_N^{N+1} \frac{\mathrm{d}x}{1+x^2}; \tag{2.19}$$

for small x,

$$\exp x - 1; \tag{2.20}$$

for large N,

$$\int_N^{N+1} \mathrm{d}x \log(1+x)\,. \tag{2.21}$$

2.7 Condition Number

The condition number, $\kappa(f, x)$, of a function $f(x)$ quantifies the maximum relative rate of change of a function with respect to changes in its arguments. Functions with a low condition number are said to be well-conditioned, and those with a high condition number, ill-conditioned. The outputs of well-conditioned functions are less sensitive to small changes to their inputs arising from, *e.g.*, measurement error or approximation errors. Ill-conditioned functions, by contrast, may have outputs that vary strongly on small changes to their inputs, inducing errors in the output disproportionate to those in the input.

Mathematically, the condition number of a function $f(x)$ is given by,

$$\kappa(f, x) = \lim_{\epsilon \to 0} \sup_{\|\delta x\| \leq \epsilon} \frac{\|\delta f(x)\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|}. \tag{2.22}$$

For differentiable functions of one variable, this expression simplifies to,

$$\kappa(f, x) = \left| \frac{x f'(x)}{f(x)} \right|, \tag{2.23}$$

which the keen-eyed will recognise as the logarithmic derivative of f divided by the logarithmic derivative of x,

$$\left| \kappa(f, x) = \frac{\mathrm{d}}{\mathrm{d}x} \log f \Big/ \frac{\mathrm{d}}{\mathrm{d}x} \log x \right|. \tag{2.24}$$

The logarithm of the condition number gives an approximate indication of how many digits of accuracy will be lost in addition to those lost through imprecision in approximation.

For functions of many variables we can write,

$$\frac{\|J(\mathbf{x})\|}{\|f(\mathbf{x})\| / \|\mathbf{x}\|}, \tag{2.25}$$

where $J(\mathbf{x})$ denotes the Jacobian matrix of partial derivatives of f at $\mathbf{x}$. Notice that this expresses the relative rate of change of a function with respect to the relative rate of change of $\mathbf{x}$, **the norm of a vector containing its arguments**, not just the relative rate of change of an individual argument.

Info: The ℓ_p vector norm $(p \geq 1)$ of a vector $\mathbf{x} = (x_1, \ldots, x_n)$ is

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}. \tag{2.26}$$

The 2-norm is the familiar Euclidean norm; the set of n-dimensional vectors with the same 2-norm covers the surface of a n-1-dimensional hypersphere; the set of vectors with the same 1-norm, called the taxicab or Manhattan norm, forms the surface of a cross polytope (*e.g.*, in 3 dimensions, the surface of the octahedron).

As p approaches ∞, l_p approaches l_∞,called the infinity, or maximum norm,

$$\|x\|_\infty = \max(|x_1|, \ldots, |x_n|). \tag{2.27}$$

The set of vectors with ∞-norm equal to some constant, c, covers the surface of a hypercube with side length $2c$.

2.7.1 Elementary Functions

We will now apply these results to some elementary functions. In the case of scalar multiplication, $f(x) = ax$, the condition number is,

$$\begin{aligned} \kappa(f, x) &= \left| \frac{x \cdot f'(x)}{f(x)} \right| \\ &= \frac{x \cdot a}{ax} \\ &= 1, \end{aligned} \tag{2.28}$$

norm	ℓ_1	ℓ_2	ℓ_∞
$\kappa(+,x,y)$	$2\frac{\lvert x\rvert+\lvert y\rvert}{\lvert x+y\rvert}$	$\frac{\sqrt{2(x^2+y^2)}}{\lvert x+y\rvert}$	$\max\left(\left\lvert\frac{x}{x+y}\right\rvert,\left\lvert\frac{x}{x+y}\right\rvert\right)$
$\kappa(\times,x,y)$	$\lvert x/y\rvert+\lvert y/x\rvert+2$	$\lvert x/y\rvert+\lvert y/x\rvert$	$\max(\lvert x/y\rvert,\lvert y/x\rvert)$

Table 2.2: Condition number of addition and multiplication for three choices of vector norm.

which suggests that scalar multiplication is always well-conditioned, so the accuracy of the output $f(x)$ will be the same as the accuracy in the input, x.

Multiplication of two variables, $f(x,y)=xy$, has distinctly different characteristics,

$$\begin{aligned}\kappa(f,x,y) &= \frac{\left\|\left(\frac{\partial f}{\partial x},\frac{\partial f}{\partial y}\right)\right\|}{\|f(x,y)\|/\|(x,y)\|}\\ &= \frac{\|(x,y)\|^2}{|(x\cdot y)|}\end{aligned} \tag{2.29}$$

In general, the condition number will differ for different choices of norm. Table 2.2 shows the condition number evaluated using the ℓ_p-norm for $p=1,2,\infty$. For all choices of norm, multiplication of two numbers x,y is well-conditioned (κ is close to 1) when the numbers are similar in magnitude, but becomes ill-conditioned when either $x\gg y$ or $y\gg x$.

It is important to note that the condition number does not tell us what the error in f will be; it simply gives an upper bound on the ratio of the relative error in f and the relative error in $\mathbf{x}$. In the case of multiplying two numbers: take $\mathbf{x}=(a,b)$ and $\delta\mathbf{x}=(\delta a,\delta b)$. Let . Consider the case $(a,b)=(1,10000)\,;\|\delta x\|=1$. The error in x could be mostly in a, mostly in b, or spread between the two.

If the error is entirely in b, then the error in the product is 1; *i.e.*, the same as in the error in the input. If, however, the error is entirely in a, the error in the product is 10000, four orders of magnitude greater than the error in the input! This magnification of error is just below the condition number of f at $\mathbf{x}$.

Figure 2.3 shows histograms of ratios of relative errors for a range of a,b, for random samples of $\delta\mathbf{x}$ with $\|\delta x\|=1$. The histograms are bounded from the right by κ, and are strongly skewed toward κ. As expected, the error in the output is smallest most often when $a=b$.

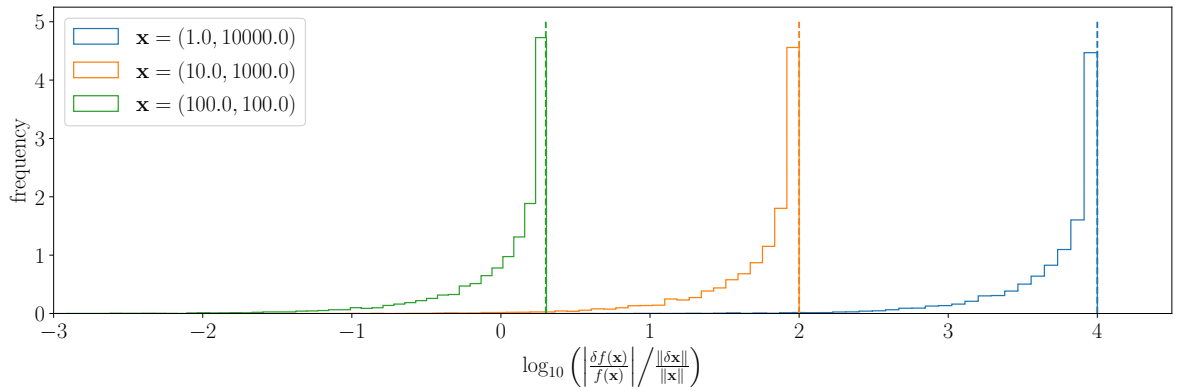

Figure 2.3: Histograms of the logarithm of the ratio of the relative error in $a\cdot b$ and the relative error in $\mathbf{x}=(a,b)$ for a sample of 10,000 error vectors, $\delta\mathbf{x}$, with elements sampled from the standard normal distribution. When $a=b$, the ratio is smallest.

For scalar addition, $f(x) = a + x$, the condition number is,

$$\begin{aligned} \kappa(f, x) &= \left| \frac{x \cdot f'(x)}{f(x)} \right| \\ &= \left| \frac{x}{a + x} \right|. \end{aligned} \tag{2.30}$$

Addition is well-conditioned except near the point $a + x = 0$, *i.e.*, $x = -a$; at this point, the true value of f is exactly 0, and any deviation of x from its true value produces a totally inaccurate result. Addition of two variables, $f(x, y) = x + y$ is also ill-conditioned around $x = -y$; expressions for κ are listed in 2.2.

Referring back to our example of the discriminant of the quadratic equation,

$$ax^2 + bx + c = 0 \rightarrow x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2.16}$$

Fixing the value $a = 1/4$, the expression simplifies to,

$$f(b, c) = 2\left[-b \pm \sqrt{b^2 - c}\right] \tag{2.31}$$

Taking derivatives and plugging in to 2.24, we find,

$$\kappa(f, b, c) = \frac{\sqrt{f^2 + 1}}{f} \cdot \sqrt{\frac{b^2 + c^2}{b^2 - c}}, \tag{2.32}$$

indicating that the discriminant is ill-conditioned not only when $c \ll b^2$, but also when $c \approx b^2$. As c goes to zero, one of the solutions comes close to zero, and either side of $c = b^2$, the number of real solutions changes; both cases constitute catastrophic cancellation resulting from ill-conditioned addition.

Although the condition number of a monomial x^p is $|p|$, that of a polynomial depends on the coefficients, it is difficult to find the roots of ill-conditioned polynomials. Data fitted to an exponential function, $\exp x$, may be pleasing to the eye, but one should be careful to note that its condition number grows as $|x|$, *i.e.*, the number of decimal digits lost grows as $\log(|x|)$, becoming large for both large, negative, and large, positive x.

Bibliography