Lecture notes on Python programming

Amit Moscovich
Tel-Aviv University
https://mosco.github.io/

2026-01-15

Contents

Preface

These lecture notes contain much of the material covered in the class "Introduction to Computers for Statisticians", taught at Tel-Aviv University 2022–2025.

Who is this for? In addition to accompanying the course, these notes may be helpful to anyone who would like to analyze data using Python. They can also serve as a crash course for data scientists proficient in other languages such as R or MATLAB. No prior programming experience is assumed.

What's so special about these notes? There are many good introductions to Python programming, but most of them are aimed at aspiring programmers who wish to build full applications. In contrast, these notes focus on *interactive data analysis* using the IPython interpreter. They include many short, runnable examples to elucidate key concepts. I favor this approach because it:

- directly serves the needs of data scientists and analysts who spend much of their time exploring data in an interactive shell.
- encourages tinkering and experimentation, leading to a rapid learning cycle.
- allows readers to follow along by testing and tweaking the examples on their computers.

[concept name] Whenever a new concept is introduced, its name appears in brackets on the left margin.

Mistakes? Suggestions? Cute examples?, please write me: **mosco@tauex.tau.ac.il**

– Amit Moscovich, Tel Aviv University. https://mosco.github.io/

These students helped improve this text by finding bugs: Amir Arzi, Keshet Rosenberg, Noa Taganya.

Part I

Python basics

Chapter 1. Setting up Python

Important note

Follow these installation steps before you start reading the book!

Step 1: Install the Anaconda Python distribution. This distribution includes the Python interpreter, the IPython shell and hundreds of Python packages. Download and install it from here: https://www.anaconda.com

Step 2: Open a terminal.

- **Windows users:** click start, search for Anaconda Powershell Prompt, then click to open.
- **Mac users:** Run Terminal.app. You can use Spotlight Search to find it. After it's running, right-click on its icon and press Options→Keep in Dock to make it easily accessible in the future.
- **Linux users:** Open "Terminal".

[interpreter]

Step 3: Run `ipython`. Type the command **`ipython`** inside your terminal window and press *Enter*. You should see something like this:

```
Python 3.12.2 (main, Feb 16 2024, 20:54:21)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.27.0 -- An enhanced Interactive Python. Type '?' for help.
>>>
```

Congratulations! You now have a running *Python interpreter*. Every time you type a command and press *Enter*, the Python interpreter reads your command and runs it. Keep this window open as you read the book so that you could experiment as you study.

Chapter 2. First steps

Let's check if Python can add.

```
Python 3.12.2 (main, Feb 16 2024, 20:54:21)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.27.0 -- An enhanced Interactive Python. Type '?' for help.
>>> 1+1
2
```

[numeric ops] Yes it can! Python can do anything your calculator can and it respects the usual order of operations:

```
>>> 1+2*3
7
>>> (1+2)*3
9
>>> ((1+2)*3)*(3-1)
18
```

[power] To raise a number to a power, use the `**` operator. For example, 3^2 is given by

```
>>> 3**2
9
>>> -3**2     # equals -(3**2)
-9
>>> (-3)**2
9
```

[long] Python supports arbitrarily large numbers,

```
>>> 2**400
258224987808690858965591917200301187432970579282922351283065935654064762201684119462964535328013783143590317197274749337
```

[float division] Let's try division:

```
>>> 3/2
1.5
>>> 4/2
2.0
```

[int, float] The result of the last division is 2, but we got 2.0. Why? Python supports two different *types* of numbers: **`int`** (integer) and **`float`** (floating-point) which holds numbers with a decimal point. Adding or multiplying two integers together always results in an integer but dividing two integers can result in non-integer values. For consistency, the division operator in Python always returns a number of type **`float`**.

[type] We can query the type of a result:

```
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
```

Each type has a set of allowed operations.

[int division] To perform *integer division*, use the `//` operator, that rounds down:

```
>>> 4//2
2
>>> 3//2
1
```

[modulus] The modulus operator `%` gives the remainder of the division:

```
>>> 3%2
1
```

Example 2. We can use `//` and `%` to extract the decimal digits of a number.

```
>>> 365 % 10
5
>>> 365 // 10
36
>>> (365 // 10) % 10
6
>>> 365 // 100
3
```

2.1 Variables

Example 3. What is the surface area of a pizza of diameter 15cm? How about 25cm? Recall the famous formula for the area of a pizza: $A = \pi r^2$ where $\pi = 3.1415...$ and r is the radius (half of the diameter).[1]

```
>>> 3.141593 * (15/2)**2
176.71460625
>>> 3.141593 * (25/2)**2
490.87390625
```

[variables] Rather than repeating ourselves, we can assign the value π to a variable.

```
>>> pi = 3.14159265358979
>>> pi * (15/2)**2
176.7145867644257
>>> pi * (25/2)**2
490.8738521234047
```

This assignment operator `=` binds a name `pi` to the number on the right-hand-side. Using variables makes the code more readable and more consistent. It also involves less typing and reduces the chance of errors. When you assign a value to a variable, Python does not print the value. e.g.

```
>>> area = pi * (15/2)**2
```

You can reassign variables, changing their value or even their type:

```
>>> x = 10
>>> x = 11.5
>>> x
11.5
```

[del] You can delete variables. Accessing a deleted variable results in a **`NameError`**.

[1] Amazingly, this formula was known to the ancient Greeks 2000 years before the invention of pizza!

```
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

To increment **x** by one we can use an extra variable:

```
>>> y = x+1
>>> x = y
```

Here's an easier way to increment:

```
>>> x = x+1
```

The expression **x=x+1** first evaluates the right expression **x+1** and then assigns that value to **x**. Python has a special syntax for this:

```
>>> x = 10
>>> x += 3
>>> x
13
```

The expression **x += k** is just a shorthand for **x = x + k**. There are similar shorthands for multiplication, division, etc.

```
>>> zz = 10
>>> zz *= 2
>>> zz
20
```

Question. What do the following commands do?

```
>>> b = 5
>>> b += b+b
```

Try it yourself in the Python interpreter! When using the plus-equals operator **+=**, first the right-hand-side is evaluated and then this value is assigned to the left-hand-side.

[underscore] The special variable underscore **"_"** always keeps the result of the last command:

```
>>> 1+1
2
>>> _*2
4
>>> _*2
8
```

[dir] To list all the variables use the **dir()** command:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
    '__package__', '__spec__', 'pi', 'x', 'zz', 'b']
```

The variables that start and end with double underscores like **__name__** have special roles. You will learn about some of them later. *Warning!* When you close the Python interpreter, the variables in the current session are gone forever. If you want to keep them for later you will need to save them to a file as we will learn in Chapter 15.

2.2 The Python standard library

Python has many in modules. For a full list, see the official docs:
https://docs.python.org/3/library/

[import math] One example is the **math** module. To load it we use the **import** command:

```
>>> import math
>>> math.pi
3.141592653589793
```

[namespace]

When we import **math**, Python loads various functions and variables into the **math** *namespace*. To see all the imported names you can use **dir(math)**:

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
    'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
    'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf',
    'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
    'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
    'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
    'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
    'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
    'trunc', 'ulp']
```

The elements in the **math** namespace can be accessed as **math.pi**, **math.sin**, etc. Just by looking at the output of **dir(math)** we can't tell that **math.pi** is a number and that **math.sin** is a function. We can use the interpreter to check:

```
>>> math.pi
3.141592653589793
>>> math.sin
<built-in function sin>
```

You call a function by writing its name followed by parentheses containing the input arguments:

```
>>> math.sin(0)
0.0
>>> math.cos(0)
1.0
```

Question. Can you guess what **math.sin(math.pi)** will do?

```
>>> math.sin(math.pi)
1.2246467991473532e-16
```

This notation means the result is $1.2246467991473532 \times 10^{-16}$. So it is almost, but not quite, zero. *But we know that* $\sin(\pi) = 0$*, so why isn't it exactly zero!?* Unfortunately, computers cannot store numbers such as π exactly. The value of the *float* variable **math.pi** is a fraction that approximates π. Furthermore, in general **math.sin(x)** can only guarantee to return an approximation of $\sin(x)$.

[help]

We can type **help(math)** to get some information.

```
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.9/library/math

    The following documentation is automatically generated from the
    Python source files.  It may be incomplete, incorrect or
    include features that are considered implementation detail and
    may vary between Python implementations.  When in doubt,
    consult the module reference at the location listed above.

DESCRIPTION
```

```
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.
.
.
.
    trunc(x, /)
        Truncates the Real x to the nearest Integral toward 0.

        Uses the __trunc__ magic method.

    ulp(x, /)
        Return the value of the least significant bit of the float x.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    /opt/anaconda3/lib/python3.9/lib-dynload/math.cpython-39-darwin.so

(END)
```

Chapter 3. Strings

So far we have seen the **int** and **float** data types. Now we will learn about **string**.

```
>>> s = 'Hello, world!'
>>> s
'Hello, world!'
>>> type(s)
str
```

[print] When we evaluate **s** we get the Python representation of the string. To print the string without the surrounding quotes we can use the **print** function:

```
>>> print(s)
Hello, world!
```

What if we want to have single quotes **'** inside our string? A string like **'Let's go'** would not work, since Python would read this as the string **'Let'** and then be confused by the **s go'**. Instead of single quotes, we can use double-quotes to delimit the string. A double-quoted string can have single-quotes:

```
>>> print("'")
'
```

Similarly, we can use double-quotes **"** inside single-quoted strings:

```
>>> print('"')
"
```

[escaping] What if we want both single and double-quotes in the same string? We can prefix the quotes by \ as follows:

```
>>> print('Single quote: \' Double quote: \" ')
Single quote: ' Double quote: "
```

In programming jargon, this is known as *string escaping* and the backslash character \ is known as the *escape character*. Sequences such as **\'** and **\"** inside a string are called *escape sequences*. Other interesting escape sequences include **\n** (newline), **\a** (rings a bell!) and \\ (guess what this does!). Let's try to use the newline character:

```
>>> s = 'Blah blah blah.\nBlah blah.\nBlah'
>>> print(s)
Blah blah blah.
Blah blah.
Blah
```

[triple quotes] There's a nicer way to include new lines using Python's triple-quotes, which works on multiple lines and treats newline as **\n**:

```
>>> s = '''Blah blah.
... blah.
... '''
>>> print(s)
Blah blah.
```

```
blah.

>>> s
'Blah blah.\nblah.\n'
```

3.1 Adding and printing strings

[string +] Strings can be added together. Let's say hello:

```
>>> s0 = "Hello "
>>> s1 = "Mike"
>>> print(s0+s1)
Hello Mike
```

The **+** operator behaves very differently on strings and numbers! When we apply **+** to two numbers of type **int** or **float** it performs numeric addition but when applied to strings it performs string concatenation. What if we try to add an **int** and a **str** ?

```
>>> s2 = 6
>>> print(s0+s2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

This fails with a **TypeError** and an informative error message: strings can only be added to strings. A **TypeError** is one of Python's built-in exceptions that happens whenever there is a type mismatch. We will learn more about exceptions later.

[str/int conv.] To fix the code, we can convert the int **6** to a string using **str(6)**:

```
>>> print(s0+str(6))
Hello 6
```

[arguments] Another way is to give multiple arguments to **print**. It automatically converts every argument to string and concatenates them together with spaces in between:

```
>>> print(s0, s2)
Hello  6
```

Note the double space! We don't need it since **"Hello "** already has a trailing space. Let's use the **help** function to understand how to get rid of the space:

```
>>> help(print)
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

[function signature] The line **print(*args, sep=' ', end='\n', file=None, flush=False)** is the *function signature* that tells us which arguments the function takes. It shows that

[keyword arg.] **print** takes a variable number of regular (positional) arguments. It also takes 4 *keyword arguments* with default values. Let's go through them in detail:

- `*args` means that `print` takes a variable number of positional arguments. e.g. both `print('word')` and `print('two', 'words')` work.
- The `sep=' '` in the function signature means that the function takes an optional argument named `sep` whose default value is `' '`. It can be overriden.
- The argument `end='\n'` means that, by default, `print` adds a newline at the end.
- Don't worry about `file` and `flush`. These are used when `print` is used to write to a file.

Let's try overriding the `sep` parameter:

```
>>> print(s0, s2, sep=':')
Hello :6
```

We can remove it altogether by choosing an empty string as separator:

```
>>> help(print)
>>> print(s0, s2, sep='')
Hello 6
```

Converting from `str` to `int` also works... But only when it is sensible.

```
>>> int('6')
6
>>> int('-89739087362893726')
-89739087362893726
>>> int('5.3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '5.3'
>>> int('haha')
Traceback (most recent call last)
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'haha'
```

3.2 f-strings

We often want to print combinations of text strings and variables, for example:

```
>>> str(hours) + ':' + str(minutes) + ':' + str(seconds)
'17:23:43'
>>> hours = 17
>>> minutes = 23
>>> seconds = 43
>>> str(hours) + ':' + str(minutes) + ':' + str(seconds)
'17:23:43'
```

[f-strings] *f-strings* are a more convenient way to do this:

```
>>> f'{hours}:{minutes}:{seconds}'
'17:23:43'
```

You can place any Python expression inside the braces:

```
>>> filenum = 5
>>> n_files = 23
>>> print(f'Reading file {filenum} out of {n_files}. {n_files-filenum} more
    to go...')
Reading file 5 out of 23. 18 more to go...
```

f-strings have a special syntax for number padding:

```
>>> hours = 5
>>> minutes = 23
>>> seconds = 7
>>> print(f'{hours}:{minutes}:{seconds}')
5:23:7
>>> print(f'{hours:2}:{minutes:02}:{seconds:02}')
 5:23:07
```

3.3 String indexing and len

[indexing] Individual characters can be accessed using **`string[i]`** notation where **`i`** is the index of the character we want. Note that indexing starts at zero!

```
>>> abc = 'abcdefghijklmnopqrstuvwxyz'
>>> abc[0]
'a'
>>> abc[1]
'b'
>>> abc[25]
'z'
>>> type(_)
str
```

[len] Python doesn't have a special type for single characters, so fetching a single character returns a string of length one. The **`len`** function gives the length of the string:

```
>>> len(abc)
26
```

We can use it to access the last letters of a string:

```
>>> abc[len(abc)-1]
'z'
>>> abc[len(abc)-2]
'y'
```

This can be a little cumbersome so there is a shorthand using negative indices:

```
>>> abc[-1]
'z'
>>> abc[-2]
'y'
```

The valid indices of a string **`s`** are: **`s[0], ..., s[len(s)-1]`**. What happens if we access an index out of the valid range?

```
>>> abc[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Phew! In low-level languages like C, a bad index could result in all sorts of crazy and unexpected behaviors. It could change another variable, cause the program to crash or (on older systems) make the computer restart! Python always checks that the index is valid, so if the index is out of range we get a nice error message.

3.4 String slicing

[string slicing] We can fetch ranges of characters using *slice notation*:

```
>>> abc[1:3]
'bc'
>>> abc[1:-1]
'bcdefghijklmnopqrstuvwxy'
>>> abc[0:10:2]
'acegi'
```

The general syntax of a slice is **s[start:end:step]** and it is not inclusive of the end index. i.e. the slice **s[start:end]** includes the indices **start, start+1, ..., end-1**. You don't have to specify all parts of a slice since there are defaults for all values: **start=0**, **end=len(s)**, **step=1**.

```
>>> abc[3:]
'defghijklmnopqrstuvwxyz'
>>> abc[:-1]
'abcdefghijklmnopqrstuvwxy'
>>> abc[:10:2]    # Return characters 0...9 in steps of 2
'acegi'
>>> abc[::2]      # Return all characters in even indices
'acegikmoqsuwy'
```

Negative step sizes can also be used:

```
>>> abc[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

The use of negative step sizes can be confusing and requires swapping the start and end.

```
>>> abc[10:0:-1]
'kjihgfedcb'
```

Note that the letter **'a'** is not included in this slice. We can get it like this

```
>>> abc[10::-1]
'kjihgfedcba'
```

It is best to avoid slices with negative steps, except for the simple case **s[::-1]**.

3.5 String methods

Strings come equipped with a few built-in functions (called *string methods*). You can use **dir()** to list them.

```
>>> s = 'abc'
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
    '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
    '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
    '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
    '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
    '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
    'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
    'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
    'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
    'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
    'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace',
    'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
    'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
    'upper', 'zfill']
```

[help] Let's use the **help** function to get more information about these functions,

```
>>> help(s.upper)
Help on built-in function upper:

upper() method of builtins.str instance
    Return a copy of the string converted to uppercase.
```

This function does not accept any parameters, so we call it like this

```
>>> s.lower()
'abc'
```

Some of the string methods are used to test propertis of the string

```
>>> s.islower()
True
>>> s.isupper()
False
```

See Python's documentation for the full list of string methods:
https://docs.python.org/3/library/stdtypes.html#string-methods

3.6 Unicode

[unicode]

Python strings support unicode. This means you can use use strings in any language. Even Elon Musk's kid would one day be able to use Python to print his own name. Try typing **`print('X \u00C6 A-12 Musk')`** in the Python interpreter. You can even copy&paste emojis!. For emojis, an alternative is to use *CLDR short names*. For example, try typing **`'\N{face with tears of joy}'`**. Here is a list of emoji CLDR short names:
https://unicode.org/emoji/charts-13.0/emoji-list.html
Unfortunately, not all of them are supported.

3.7 The IPython shell

The IPython shell provides a nicer experience than the default python shell. To use it simply type **`ipython`** instead of **`python`**. Aside from the colors, here are some things supported by IPython:

1. In IPython, each output line is numbered. You can use values from history by typing underscore followed by the output number, for example **`_5`**.
2. **`%whos`** to list all the variables
3. Copying&pasting code just works, even with all the IPython line numbers, etc.
4. You can save the code typed in lines 1-4 type this: %save testing.py 1-4
5. Shell commands: any command that starts with ! is passed to the shell/command line. e.g. **`!cd myworkdir`**.
6. This nifty command creates a publicly available web page with your code in selected lines: **`%pastebin -d 'title' 1-5`**

There are tons of other IPython magic commands. Type **`%magic`** and let me know if you find anything useful!

Chapter 4. Conditions

To test for equality and inequality, we can't use the **=** operator since it is reserved for assignment. Instead, we use the **==** operator to test for equality and the **!=** operator to test for inequality.

[==, !=]

```
>>> 6 == 6
True
>>> 6 == 7
False
>>> 6 != 7
True
```

The result of a comparison operator is always of **bool** type (short for boolean). Booleans are either **True** or **False**. They can be assigned to variables just like any other value:

```
>>> result = (1+1 == 2)
>>> type(result)
bool
>>> result
True
```

The parentheses around **1+1 == 2** are not required but good for readability. We can compare numeric values using the operators **<, >, <=, >=**.

```
>>> x = 10
>>> x < 10
False
>>> x <= 10
True
>>> type(_)
<class 'bool'>
```

Python's comparison operators are listed in the following table:

operator	*meaning*	*supported types*
==	equal	int, float, str, bool, ...
!=	not equal	int, float, str, bool, ...
<	smaller than	int, float
<=	smaller or equal	int, float
>	larger than	int, float
>=	larger or equal	int, float

4.1 Logical operations

Python is equipped with the following logical (aka boolean) operators: **and**, **or**, **not**. They can be used to define complicated conditions and work exactly as you'd expect:

```
>>> False or True
True
>>> False and True
False
>>> True and True
True
>>> not True
False
>>> not False
True
>>> True and not False
True
```

Example 4. Here are 3 equivalent ways to check if **percent** is in the range $[0, 100]$:

```
is_invalid_percent = (percent<0) or (percent>100)
is_invalid_percent = not ((percent>=0) and (percent<=100))
```

[precendence]

Since Python's logical operators have a lower *operator precedence* than the comparison operators, we can drop some parentheses:

```
is_invalid_percent = percent<0 or percent>100
is_invalid_percent = not (percent>=0 and percent<=100)
```

4.1.1 Chained comparisons

[chained comparisons]

Python supports *chained comparisons operators* as follows

```
>>> 5 < x <= 10
True
```

This is equivalent to testing that both **5 < x** and **x <= 10**. Be careful not to put parentheses when you chain!

```
>>> 5 < (x <= 10)
False
```

What's going on here? The comparison **x <= 10** is performed first and it returns **True**. Then Python evaluates the expression **5 < True** – an **int** to **bool** comparison! When comparing a **bool** to an **int**, Python performs an automatic conversion as follows: **False** $\rightarrow$ **0** and **True** $\rightarrow$ **1**. So the comparison **5 < (x <= 10)** reduces to the comparison **5 < 1** which does not hold.

4.2 If-elif-else statements

[if]

The **if** statement is used when we want to conditionally run a piece of code. It is used like this: Here is an example of code from a train controller program:

```
>>> MAX_SPEED = 160
>>> if speed > MAX_SPEED:
...     print(f'Can only go up to {MAX_SPEED} km/h')
...     speed = MAX_SPEED
```

[code block]

If the logical condition **speed > MAX_SPEED** is satisfied then lines 3-4 in the indented block that follows it will be executed. In any case **set_target_speed(speed)** will be called. Python expects an indented *code block* immediately after **if <condition>:** line (note the colon at the end of the **if** statement). The end of the code block is marked by the end of the indentation.

Note on code blocks.
Unlike other languages which use notation like **{}** for code blocks, Python simply uses text indentation. Any number of spaces can be used for indentation but 4 is customary. You can use a single tab instead, however, mixing tabs and spaces is not allowed. Most Python-aware text editors will convert each tab to spaces.

Let's write code that tells us if a number is even or odd

```
>>> if number%2 == 0:
...     print('Even')
... if number%2 != 0:
...     print('Odd')
```

[if-else] Instead of repeating the calculation, we can use an if-else statement,

```
>>> if number%2 == 0:
...     print('Even')
... else:
...     print('Odd')
```

This form is easier to read, less error-prone and faster to run. You can nest if statements within if statements using indentation.

Example. Single-ride ticket prices for the Jerusalem light-rail are set as follows:

Ticket type	**Single-ride price** (ILS)
Child up to 5	0
Elderly above 75	0
Senior citizen	2.25
Everybody else	5.5

Here's an implementation of this logic using **`if-else`** statements:

```
>>> FULL_PRICE = 5.5
>>> if age <= 5:
...     price = 0
... else:
...     if age >= 75:
...         price = 0
...     else:
...         if is_senior_citizen:
...             price = 0.5*FULL_PRICE
...         else:
...             price = FULL_PRICE
>>> print(f'Ticket price: {price} ILS')
```

Because Python uses indentation for nesting, such an **`if-else`** chain results in deeply-
[elif] nested code. The **`elif`** keyword was invented to resolve this issue.

```
>>> if age <= 5:
...     price = 0
... elif age >= 75:
...     price = 0
... elif is_senior_citizen:
...     price = 0.5*FULL_PRICE
... else:
...     price = FULL_PRICE
```

Note Such an if-elif-else chain makes it very clear that exactly *one* of the blocks is executed. The condition of an **`if`** statement does not have to be a boolean, the condition is automatically converted to a boolean. Here are some examples.

```
>>> if 0:
...     print('Zero')
>>> if 1:
...     print('One')
One
>>> if '':
...     print('Empty string')
>>> if 'bee':
...     print('BZZZZZZZ')
BZZZZZZZ
```

This can be a little confusing so it's generally recommended to explicitly test for equality (e.g. use **`if s == ''`** instead of **`if s`**).

Chapter 5. Lists

[list]

Python lists are used to hold ordered collections of arbitrary type. Let's see an example using the last 5 years in which general elections were held in Israel.

```
>>> election_years = [2022, 2021, 2020, 2019, 2019]
>>> len(election_years)
5
>>> election_years[1]
2021
>>> election_years[:2]
[2022, 2021]
>>> election_years[-1]
2019
```

[list editing]

Lists can be indexed and sliced just like strings. Unlike strings, they can also be modified.

```
>>> election_years[0] = 'Vote!'
>>> election_years
['Vote!', 2021, 2020, 2019, 2019]
```

You can even use assignment on a whole slice:

```
>>> election_years[2:] = ['bibi', 'bibi', 'bibi']
>>> election_years
['Vote!', 2021, 'bibi', 'bibi', 'bibi']
```

[list.append]

As you can see, the elements of a list don't all have to be of the same type. Lists are often constructed by starting with an empty list and calling the **`append()`** method.

```
>>> primes = []
>>> primes.append(2)
>>> primes.append(3)
>>> primes.append(5)
>>> primes
[2, 3, 5]
```

[list +/*]

Lists can be added together using the **`+`** operator and repeated using the **`*`** operator.

```
>>> ['a','b'] + ['c','d']
['a', 'b', 'c', 'd']
>>> ['a','b']*3
['a', 'b', 'a', 'b', 'a', 'b']
```

Rather than starting with an empty list, it is also common to initialize a list from a base value and then edit it. For example, we can track course submissions as follows,

```
>>> whosubmitted = [False]*23
>>> whosubmitted[12] = True
>>> whosubmitted[17] = True
```

[list methods]

Some other things we can do with lists:

```
>>> lis = [2,5,8]
>>> lis.insert(1, -100)
>>> lis
[2, -100, 5, 8]
>>> lis.reverse()
>>> lis
[8, 5, -100, 2]
>>> lis.sort()
>>> lis
[-100, 2, 5, 8]
```

There are several other list methods which you may find useful. Use **help(list)** or **dir(list)** to check them out.

5.1 Mutability

We can assign the same list to several variables, but this can result in surprising behavior.

```
>>> a = [1,2]
>>> b = a
>>> b[1] = 99
>>> a
[1, 99]
>>> b
[1, 99]
```

[references]

What's going on here? In Python, variables are *references* to an object. The initialization **a = [1,2]** creates a new list object and then makes the variable name **a** refer to it. The assignment **b=a** copies the reference but not the list. The reason we hadn't encountered this behavior before is that lists are *mutable* objects, which means they can be modified. In contrast, numbers and strings are *immutable*. They cannot be changed, only created and destroyed.

[id]

To get a little more clarity we can use the **id** function. It gives a unique identifier of the object that a reference points to.

```
>>> a = [1,2]
>>> b = [1,2]
>>> c = a
>>> id(a)
140200520310528
>>> id(b)
140200519128320
>>> id(c)
140200520310528
```

In fact, the value returned by **id** is just the memory address where the object is stored.

5.2 Copying

To make a copy of a list we can initialize a new empty list and copy the references from the old list.

```
>>> a = [1,2]
>>> b = [-1]*2
>>> b[0] = a[0]
>>> b[1] = a[1]
>>> a
```

```
[1, 2]
>>> b
[1, 2]
>>> b[1] = 99
>>> a
[1, 2]
>>> b
[1, 99]
```

[shallow copy]

This element-by-element copying operation is implemented in the **list.copy** method and also in **copy.copy** (the **copy** function of the **copy** module).

```
>>> a = [1,2]
>>> b = a.copy()
>>> b[1] = 99
>>> a
[1, 2]
>>> b
[1, 99]
```

This type of copy operation is known as a *shallow copy*. Since the references inside the list are copied, but the underlying objects are not.

```
>>> a = [1,[2]]
>>> b = a.copy()
>>> b[0] = 99
>>> b[1].append(222)
>>> a
[1, [2, 222]]
>>> b
[99, [2, 222]]
```

To create a full copy of a list (including sub-lists, sub-sub-lists, etc.) we can use the [deep copy] **copy.deepcopy** function.

```
>>> a = [1,[2]]
>>> import copy
>>> b = copy.deepcopy(a)
>>> b[0] = 99
>>> b[1].append(222)
>>> a
[1, [2]]
>>> b
[99, [2, 222]]
```

It works for all the Python built-in types, including many we haven't learned about yet.

5.3 Lists vs. strings

[list/str conv.] Converting a string to a list gives us the list of characters.

```
>>> s = 'abc'
>>> list(s)
['a', 'b', 'c']
>>> str(_)
"['a', 'b', 'c']"
```

Question. Why doesn't the last conversion give us the string **"abc"** back?

The **str.split** and **str.join** methods are used to split and rejoin substrings separated by a given string. Let's use them to convert a european-style date to an ISO [split, join] standard date:

```
>>> s = '23/10/2022'
>>> elements = s.split('/')
>>> elements
['23', '10', '2022']
>>> elements.reverse()
>>> elements
['2022', '10', '23']
>>> '-'.join(elements)
'2022-10-23'
```

You can split using any string, not just a single character.

```
>>> 'A::B::C'.split('::')
['A', 'B', 'C']
>>> '::'.join(_)
'A::B::C'
```

Chapter 6. Loops

[for]

Computers wouldn't be very useful if we had to type every single command. Suppose we want to print a list of numbers, we can use the **for** command.

```
numbers = [2,4,5]
>>> for number in numbers:
...     print(number)
...
2
4
5
```

On this code, the Python interpreter assigns each of the list elements (in order) to the variable **number** and then runs the indented code (line 3). It works as though we ran the following commands:

```
>>> number = numbers[0]
>>> print(number)
>>> number = numbers[1]
>>> print(number)
>>> number = numbers[2]
>>> print(number)
```

Let's write a more complicated for loop that prints the numbers in a list and their sum.

```
>>> numbers = [2,4,5]
>>> number_sum = 0
>>> for number in numbers:
...     print(number)
...     number_sum += number
... print(f'Sum: {number_sum}')
...
2
4
5
Sum: 11
```

The general structure is

```
for <variable> in <iterable>:
    <code>
    .
    .
    <code>
<later code>
```

Just like **if** statements, **for** statements must end with a colon **:** followed by an indented code block. The code block in the foor loop (lines 2-5 in the example above) is normally run once per element of **iterable**, each time **variable** is set to be a reference to the next element. The end of the code block is marked by the end of the indentation.

For loops also work on strings and many other objects that can be iterated (such objects are called *iterable*). For example, we can iterate over the characters of a string.

```
>>> for c in 'abc':
...     print(c)
...
a
b
c
```

Example. Let's use what we learned to flip the case of all the characters in a string (e.g. turn 'aBc' into 'AbC').

```
>>> s = 'aBc'
>>> for c in s:
...     if c.islower():
...         c_flipped = c.upper()
...     else:
...         c_flipped = c.lower()
...     print(c_flipped)
...
A
b
C
```

Strings are *immutable*, so we cannot modify **s**. Instead we will create a new list and append the case-flipped characters to it. Finally, we will use the **str.join** method to stick the transformed characters together.

```
>>> flipped_chars = []
>>> for c in s:
...     if c.islower():
...         c_flipped = c.upper()
...     else:
...         c_flipped = c.lower()
...     flipped_chars.append(c_flipped)
...
>>> flipped_chars
['A', 'b', 'C']
>>> s_flipped = ''.join(flipped_chars)
>>> s_flipped
'AbC'
```

Just by observing the indentation and the keywords for/if/else we can tell the flow of the code: in each loop iteration either line 4 or line 6 will be executed. The **append** in line 7 will be run exactly once per iteration which means that at the end of the loop **flipped_chars** will have the same length as **s**.

6.1 range

[range] To loop over a range of numbers use the **range** command.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

It behaves as if **range(n)** returns the list **[0, 1, ..., n-1]**. In fact, in Python 2 that is exactly what it did. However, there is no need to actually construct the list and

store it in memory. In Python 3, if we want to see all the numbers in a range we need to explicitly convert it to list.

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

A common use of the range command is to traverse the indices of a list.

```
>>> words = ['boop', 'bop', 'beep']
>>> for i in range(len(words)):
...     print(f'{i}: {words[i]}')
...
0: boop
1: bop
2: beep
```

Range commands are similar to slices. We can specify a starting number and a step size.

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

Example. Write a loop that prints the numbers 1, . . . , n on separate lines. Next to each number, if the number is smaller than **k** print "too small", if the number is larger than **k** print "too big" and if the number is equal to **k** print "just right". *Solution.* Use a loop over a range, print the number (with no newline), then print one of the 3 texts using an **if-else** statement.

```
>>> n = 6
>>> k = 3
>>> for i in range(1,n+1):
...     print(i, end=': ')
...     if i < k:
...         print('too small')
...     elif i > k:
...         print('too big')
...     else:
...         print('just right')
...
1: too small
2: too small
3: just right
4: too big
5: too big
6: too big
```

6.2 Looping over lists

Example. Creating a new list from a list of the same size. Given a list **numbers**, square every number and store the results in the same order in an output list **squares**.

```
>>> numbers = [5, 6, -10, 17, 1.234]
>>> squares = []
>>> for x in numbers:
...     squares.append(x**2)
```

```
...
>>> squares
[25, 36, 100, 289, 1.522756]
```

An alternative solution, using indices, is to build an output list of the correct size, then write to its elements.

```
>>> squares = [0]*len(numbers)
>>> for i in range(len(numbers)):
...     squares[i] = numbers[i]**2
```

Example. Creating a new list from some of the elements of an input list. Given a list of strings **`stringlist`**, construct a list that contains only palindromes.

```
>>> names = ["Anna", "Igor", "Aviva", "Habibah", "Azeeza", "Grogu"]
... pnames = []
... for name in names:
...     lowercasename = name.lower()
...     if lowercasename[::-1] == lowercasename:
...         pnames.append(name)
...
>>> pnames
['Anna', 'Aviva', 'Habibah', 'Azeeza']
```

Now, let's keep all the names, but give the palindromes some decoration:

```
>>> pnames = []
... for name in names:
...     lowercasename = name.lower()
...     if lowercasename[::-1] == lowercasename:
...         pnames.append(f'-=<{name}>=-')
...     else:
...         pnames.append(name)
...
>>> pnames
['-=<Anna>=-', 'Igor', '-=<Aviva>=-', '-=<Habibah>=-', '-=<Azeeza>=-',
    'Grogu']
```

Example. Filtering a list. Given a list of coordinates given as **`[x,y]`** return a new list that contains only those coordinates whose distance from **`[cx, cy]`** is less than **`radius`**.

```
>>> COORDINATES = [[0.42, 0.615], [0.649, 0.696], [0.947, 0.698], [0, 0.9]]
>>> outlist = []
>>> cx = cy = 0.5
>>> radius = 0.5
>>> for xy in COORDINATES:
...     x = xy[0]
...     y = xy[1]
...     if (x-cx)**2 + (y-cy)**2 < radius**2:
...         outlist.append([x,y])
...
>>> outlist
[[0.42, 0.615], [0.649, 0.696], [0.947, 0.698]]
```

Example. (iterating over two lists simultaneously) Given two lists of numbers **`v`**, **`u`**, of the same length, create a new list **`sum_v_u`** of the same length that contains the elementwise sum of the inputs.

Here we cannot use a for-loop like **`for x in v`** because it would only iterate over the elements of one of the lists. To iterate over both we will iterate over their *indices* using **`for i in range(len(v))`**.

```
>>> v = [1,2,3]
>>> u = [10,100,1000]
>>> sum_u_v = []
>>> for i in range(len(v)):
...     sum_u_v.append(u[i]+v[i])
...
>>> sum_u_v
[11, 102, 1003]
```

6.3 Nested loops

Example. Let's print the multiplication table up to **n**. We will use a **for** loop to print the rows. First, let's generate all the elements of the 3rd row of a 10×10 table.

```
>>> n = 10
>>> i = 3
>>> row_i_elements = []
>>> for j in range(1,n+1):
...     row_i_elements.append(str(i*j))
...
>>> row_i_elements
['3', '6', '9', '12', '15', '18', '21', '24', '27', '30']
```

Note that **range(1,n+1)** to loop over the numbers $1, \ldots, n$. Had we used **range(n)** we would've looped over $0, \ldots, n-1$ but unfortunately the multiplication table as we learned in primary school does not start at zero. Now, we can print the row using the **join** function:

```
>>> print(' '.join(row_elements))
'3 6 9 12 15 18 21 24 27 30'
```

Note that **join** only works for strings. That is why we had to convert **i*j** to string before appending to **row_i_elements**.

All that remains is to repeat the above code for all values of **i** in $1, \ldots, n$ using a **for** loop. The result is a **for** loop inside a **for** loop, also called a *nested loop*.

```
>>> n = 10
>>> for i in range(1,n+1):
...     row_elements = []
...     for j in range(1,n+1):
...         row_elements.append(str(i*j))
...     print(' '.join(row_elements))
...
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

This looks awful because the numbers are not properly padded. We can fix it easily by padding the strings **str(i*j)** with spaces so that they are all the same length. The following string methods can be used: **str.ljust** (adjust left), **str.rjust** (adjust right), **str.center** (center text). Here's an example:

```
>>> s = '0'
>>> s.ljust(3)
'0  '
>>> s.rjust(3)
'  0'
>>> s.center(3)
' 0 '
```

f-strings also have a built-in padding mechanism, which behaves like **`str.rjust`**. You use it like this **`f'{<python expression>:<size>}'`**. Finally, we have,

```
>>> n = 10
... for i in range(1,n+1):
...     row_elements = []
...     for j in range(1,n+1):
...         row_elements.append(f'{i*j:3}')  # Pad each number to 3 spaces
...     print(' '.join(row_elements))
...
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

6.4 Building nested lists

Most of the time, code does not print the results of every calculation. Instead results are stored in memory for later processing.

Question. How can we store the multiplication table above?

Answer. Python does not have a built-in two-dimensional array type! A common solution is to use a list of lists. Here's the *wrong* way to do it:

```
>>> n = 5                                                    # BAD CODE #
>>> table = [[]]*n
>>> for i in range(1,n+1):
...     for j in range(1,n+1):
...         table[i-1].append(i*j)
>>> print(table)
[[1, 2, 3, 4, 5, 2, 4, 6, 8, 10, 3, 6, 9, 12, 15, 4, 8, 12, 16, 20, 5, 10,
    15, 20, 25], [1, 2, 3, 4, 5, 2, 4, 6, 8, 10, 3,
6, 9, 12, 15, 4, 8, 12, 16, 20, 5, 10, 15, 20, 25], [1, 2, 3, 4, 5, 2, 4, 6,
    8, 10, 3, 6, 9, 12, 15, 4, 8, 12, 16, 20, 5,
10, 15, 20, 25], [1, 2, 3, 4, 5, 2, 4, 6, 8, 10, 3, 6, 9, 12, 15, 4, 8, 12,
    16, 20, 5, 10, 15, 20, 25], [1, 2, 3, 4, 5, 2,
4, 6, 8, 10, 3, 6, 9, 12, 15, 4, 8, 12, 16, 20, 5, 10, 15, 20, 25]]
```

Note the use of **`table[i-1]`** here, because lists are indexed from zero but our range starts at 1. Let's see the result:

```
>>> print(table)
[[1, 2, 3, 4, 5, 2, 4, 6, 8, 10, 3, 6, 9, 12, 15, 4, 8, 12, 16, 20, 5, 10,
    15, 20, 25], [1, 2, 3, 4, 5, 2, 4, 6, 8, 10, 3,
6, 9, 12, 15, 4, 8, 12, 16, 20, 5, 10, 15, 20, 25], [1, 2, 3, 4, 5, 2, 4, 6,
    8, 10, 3, 6, 9, 12, 15, 4, 8, 12, 16, 20, 5,
```

```
10, 15, 20, 25], [1, 2, 3, 4, 5, 2, 4, 6, 8, 10, 3, 6, 9, 12, 15, 4, 8, 12,
    16, 20, 5, 10, 15, 20, 25], [1, 2, 3, 4, 5, 2,
4, 6, 8, 10, 3, 6, 9, 12, 15, 4, 8, 12, 16, 20, 5, 10, 15, 20, 25]]
```

What happened here? The source of the error is the initialization of the multiplication table using **[[]]*n**. This creates an empty list **[]**, then creates a list with **n** references *to the same empty list*. The append operations all operate on the same inner list, so we get **n** copies of the multiplication table. Here's a minimal example of this behavior:

```
>>> listoflists = [[]]*2
>>> listoflists
[[], []]
>>> listoflists[0].append(7)
>>> listoflists
[[7], [7]]
```

Using the **id** function, we can see that the two elements of **listoflists** are actually references to the same list:

```
>>> id(listoflists[0]) == id(listoflists[1])
True
```

[is] A simpler way to do this test is to use the **is** operator:

```
>>> listoflists[0] is listoflists[1]
True
```

Running **a is b** checks if **a** and **b** are references to the same exact object. This is different from testing equality with **==** which can be satisfied by different objects.

```
>>> a = []
>>> b = []
>>> a == b
True
>>> a is b
False
```

The correct way to create a nested list of empty lists is NOT to use the multiplication operator ***** but instead to initialize each sublist separately.

```
>>> n = 5
>>> table = []
>>> for i in range(1, n+1):
...     row = []
...     for j in range(1, n+1):
...         row.append(i*j)
...     table.append(row)
...
>>> table
[[1, 2, 3, 4, 5],
 [2, 4, 6, 8, 10],
 [3, 6, 9, 12, 15],
 [4, 8, 12, 16, 20],
 [5, 10, 15, 20, 25]]
```

Later we will learn about the NumPy module for working with n-dimensional arrays.

6.5 Loop control

6.5.1 break

[break] To break out of a loop prematurely use the **break** command.
Example. Compute the product of elements in a list, stopping when 666 is encountered.

```
>>> for x in numbers:
...     if x == 666:
...         print('GO AWAY SATAN!')
...         break
...     product *= x
```

Example. Consider the following problem: given a list of elements **seq**, find the index of the first occurrence of an element **v**.
Solution. We can loop over the indices **0,1,...,len(seq)-1**, testing each one and issuing a **break** statement when we find the element.

```
>>> seq = [0, 7, 7, 'Boop', 2]                          # BROKEN SOLUTION #
>>> v = 7
>>> for i in range(len(seq)):
...     if seq[i] == v:
...         break
...
>>> i
1
```

Notes:

1. The variable **i** lives on after the loop ends and retains whatever value it had when the loop finished.

2. We wrote this code for lists, but actually it works on any data type that contains a sequence of elements. Such types are called *Sequence* types in Python.

The code above has a critical flaw: we can't tell the difference between a missing element and an element at the last position. In both cases the result would be equal to **len(seq)-1**. We can fix this in a few different ways:

```
match_index = -1                                          # Solution 1 #
for i in range(len(seq)):
    if seq[i] == v:
        match_index = i
        break
if match_index == -1:
    print('No match found!')
else:
    print(f'Match found at {match_index}')
```

Why did we initialize **match_index** to -1? There's nothing special about this choice as any number outside of **0,1,...,len(seq)-1** would work. We could also use a string such as **"uninitialized"** which is perhaps more descriptive. Python has a special value called **None** that is often used to indicate such uninitialized values. All **None** values are references to the same **None** object. To test if a variable is None it is common to use **var is None** which checks that both are references to the same exact element. Of course, you could also check if **var==None**.

[None]

Instead of iterating over indices in **range(len(seq))** we can iterate on the elements of **seq**, but then we have to keep track of the index **i** ourselves, incrementing it in every iteration.

```
i = 0                                                     # Solution 2 #
match_index = None
for x in seq:
    if x == v:
        match_index = i
```

```
        break
    i += 1
if match_index is None:
    print('No match found!')
else:
    print(f'Match found at {match_index}')
```

6.5.2 continue

[continue]

Another loop control statement is **continue**. It jumps ahead to the next iteration. *Example.* Compute the product of all numbers in a list, except for 0, 4 and 11.

```
>>> product = 1
>>> for x in numbers:
...     if x in [0, 4, 11]:
...         print('Skipping')
...         continue
...     print('Multiplying', x)
...     product *= x
```

You can always replace a continue with an if-else. But this involves more nesting and doesn't express our intentions as clearly.

```
>>> product = 1
>>> for x in numbers:
...     if x in [0, 4, 11]:
...         print('Skipping')
...     else:
...         print('Multiplying', x)
...         product *= x
```

6.6 While loops

Question. How can we find the smallest value n such that $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq 7$? Looping over a range is not suitable here, since we don't know how many iterations we need. For this, we can use a **while** loop.

[while]

```
>>> partial_sum = 0
>>> i = 0
>>> while partial_sum < 7:
...     i += 1
...     partial_sum += 1/i
...
>>> partial_sum
7.001274097134162
>>> i
616
```

While loops check the condition and if it is True, they run the indented code block. This is repeated until the condition becomes False. Just like in for loops, you can use the **continue** and **break** statements. For example we can solve the same question while ignoring the even numbers.

```
>>> i = partial_sum = 0
>>> while partial_sum < 7:
...     i += 1
...     if i%2 == 0:
...         continue
...     partial_sum += 1/i
```

```
...
>>> partial_sum
7.000001791784266
>>> i
337607
```

Instead of having the explicit loop condition at the top. We can use an infinite loop that breaks when a condition (or several) is satisfied.

```
>>> while True:
...     i += 1
...     if i%2 == 0:
...         continue
...     partial_sum += 1/i
...     if partial_sum >= 7:
...         break
```

[input]

Example. Using the built-in **input** function, we'll write a loop that reads text lines from the user and appends them to a list until the user types "END".

```
>>> lines = []
... while True:
...     line = input()
...     if line == 'END':
...         break
...     lines.append(line)
```

Chapter 7. Functions

Programming would be incredibly inefficient if we had to write everything from scratch every time. Functions are the basic mechanism for code reuse. They are just a way to [function] give a name to a piece of code that we can call later. Let's start with an example.

```
>>> def bell():
...     print('\a')
...
>>> bell()          # Some type of beep should be heard
```

[arguments] Functions can take an input (or argument/parameter):

```
>>> def greet(name):
...     print('Hello ' + str(name))
...
>>> greet('Adam')
Hello Adam
>>> greet('Bob')
Hello Bob
```

[return] Functions can also *return* a value:

```
>>> def is_even(num):
...     return num%2 == 0
...
>>> is_even(5)
False
>>> is_even(-8)
True
```

Beginners often write the above code like this:

```
>>> def is_even(num):                                    # Newbie version #
...     if num%2 == 0:
...         return True
...     else:
...         return False
```

This is completely unnecessary. Whenever you see code of the form

```
if <condition>:
    return True
else:
    return False
```

It is better to just return the truth value of the condition.

```
return <condition>
```

Functions that don't return a value implicitly return **None**.

```
>>> x = bell()
>>> x
>>> print(x)
None
```

Example. Let's rewrite the example from Section 6.5.1 of finding the first match in a list. This is a function of two arguments: **seq** and **x**.

```
>>> def find_first(seq, x):
...     for i in range(len(seq)):
...         if seq[i] == x:
...             return i
```

Note that if no value is returned, the function returns None. Let's call our function:

```
>>> find_first([0, 1, 1, 'Boop', 2], 1)
1
>>> find_first([0, 1, 1, 'Boop', 2], 8)
```

[named arguments]

The second call returned **None** so nothing was printed.

So far we have used *positional arguments* in the function calls: Python knows that the first argument in the call to **find_first** is **seq** and the second is **x**. We can also pass arguments by name:

```
>>> find_first(seq=[0, 1, 1, 'Boop', 2], x='Boop')
3
```

Named arguments can be given in any order:

```
>>> find_first(x='Boop', seq=[0, 1, 1, 'Boop', 2])
3
```

We can even start with positional arguments and continue with named arguments, but the opposite is not true!

```
>>> find_first([0, 1, 1, 'Boop', 2], x='Boop')
3
>>> find_first(seq=[0, 1, 1, 'Boop', 2], x)
3
  File "<ipython-input-10-ac583a06ddd1>", line 1
    find_first(seq=[0, 1, 1, 'Boop', 2], x)
                                         ^
SyntaxError: positional argument follows keyword argument
```

Chapter 8. Python modules

Before you start this chapter: From this point on you will need a Python code editor. If you don't already have a favorite editor, give VSCode a try. See Section A in the appendix.

A module is just a collection of code and functions. Let's try writing a simple module named adder.py:

[comment]

```
# Adder module by Amit Moscovich

def add(x):
    return x+5
```

Let's import this into the interpreter, with a comment at the beginning

```
>>> import adder
>>> adder.add(6)
11
```

[import as] When the name of the module is long, we can give it a shorter name,

```
>>> import adder as a
>>> a.add(6)
11
```

8.1 Reloading modules

Let's update the adder module to compute **x+100** instead of **x+5**.

```
>>> adder.add(6)
11
```

Whoops. Nothing changed! The reason is that the module was **adder** not reloaded. Even if we try to import it again, it won't help:

```
>>> import adder
>>> adder.add(6)
11
```

The reason is that Python has a built-in speedup where it only loads modules once and then keeps a copy of them in memory for later imports. To tell Python we *really* want to reload the module from disk we need to use the **importlib** module:

[importlib]

```
>>> import importlib
>>> importlib.reload(adder)
<module 'adder' from '/Users/mosco/Dropbox/code/intro-comp-stat/adder.py'>
>>> adder.add(6)
106
```

Typing **importlib.reload(adder)** every time you change the file **adder.py** is quite a hassle. Thankfully, IPython has an **autoreload** feature that reloads modified modules automatically. Follow the instructions in Appendix B to configure it.

8.2 Docstrings: documenting your code

We can use the **dir** command to see all the functions and variables defined in a module.

```
>>> import adder
>>> dir(adder)
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'add']
```

We can see that the module defines a name **add**, but it doesn't say if **add** is a function, a variable, or something else. You can ignore the names that start and end with double underscores for now. They are special variables created automatically by Python.

We can type **help(adder)** to get more information:

```
Help on module adder:

NAME
    adder - # Adder module by Amit Moscovich

FUNCTIONS
    add(x)

FILE
    /Users/mosco/Dropbox/code/intro-comp-stat/adder.py
```

[docstring]

We can document our functions using a *docstring*:

```
# Adder module by Amit Moscovich

def add(x):
    "Adds a number to x and returns the result"
    return x+5

def sub(x,y):
    """Subtract a number from x and return the result.

    Using triple-quotes we can write multi-line docstrings!"""
    return x-7
```

The docstring is the string on the line that comes right after the function definition. Let's try typing **help(adder)** again:

```
Help on module adder:

NAME
    adder - # Adder module by Amit Moscovich

FUNCTIONS
    add(x)
        Adds a number to x and returns the result

    sub(x)
        Subtract a number from x and return the result.

        Using triple-quotes we can write multi-line docstrings!
```

```
FILE
    /Users/mosco/Dropbox/code/intro-comp-stat/adder.py
```

8.3 The toolbox mindset

Unlike most builders and artists that buy their tools at the shop, programmers build many of their own tools using functions and modules since it is very easy to package snippets of code into reusable functions.

So, whenever you feel like you are repeating yourself – for example, running the same couple of processing steps on different data sets – that is a good hint that you should define a new function. Over time, you will develop your own little "toolbox" for every project, thus becoming more efficient over time.

Chapter 9. Tuples

What if we want to return several values from a function? That's easy!

```
>>> def f(x):
...     return (2*x, 3*x)
...
>>> output = f(10)
>>> output
(20, 30)
```

The output of this function is an object of new type called **tuple**.

```
>>> type(output)
tuple
>>> output[0]
20
>>> output[1]
30
>>> len(output)
2
```

[tuple] Tuples are similar to lists, but are *immutable*.

```
>>> t = (1,2,3)
>>> t[1]
2
>>> t[1] = 5
TypeError                                 Traceback (most recent call last)
<ipython-input-28-c53026d12066> in <module>
----> 1 t[1] = 5

TypeError: 'tuple' object does not support item assignment
```

Let's try to make a tuple of length one:

```
>>> t = (7)
>>> t
7
>>> type(t)
int
```

Unfortunately, **(7)** is just a parenthesized number **7**. To let Python know we really mean a tuple of length one we need to use the following funny syntax:

```
>>> t = (7,)
>>> t
(7,)
>>> type(t)
tuple
```

It is easy to convert between lists and tuples:

```
>>> lis = [1,2,3]
>>> tup = tuple(lis)
```

```
>>> tup
(1, 2, 3)
>>> list(tup)
[1, 2, 3]
```

[unpacking] A nice feature of tuples is *tuple unpacking*, or multiple assignment

```
>>> t = (1,2)
>>> a,b = t
>>> tup = 1,2
>>> a,b = tup
>>> a
1
>>> b
2
```

Example. Swap the values of **a,b**:

```
>>> a = 5                                          # C-style solution #
>>> b = 3
>>> tmp = a
>>> a = b
>>> b = tmp
>>> a,b
(3, 5)
```

Using tuple unpacking we can produce a nicer solution:

```
>>> a = 5
>>> b = 3
>>> (a,b) = (b,a)
>>> a
3
>>> b
5
```

9.1 enumerate

[enumerate] To iterate over a sequence of elements and their indices use the **enumerate** function.

```
>>> for (i,x) in enumerate(['a','b','c']):
...     print(i,x)
...
0 a
1 b
2 c
```

This is just a nicer syntax than using **for i in range(len(seq))** and then working with **seq[i]**. Also it works with sequences that don't support indexing.

9.2 zip

[zip] The **zip** function is used to access matching elements from two (or more lists)

```
>>> lis1 = [1,2,3]
>>> lis2 = ['a','b','c']
>>> for (a,b) in zip(lis1,lis2):
...     print(a,b)
...
1 a
2 b
3 c
```

Note that **zip** does not create a list of pairs but behaves like one in the sense that we can iterate over it using a for loop. Such objects are called *Iterators*.

```
>>> zip(lis1,lis2)
<zip at 0x7f78e06ff140>
```

zip can also be used with more than two arguments:

```
>>> lis1 = [1,2,3]
>>> lis2 = ['a','b','c']
>>> lis3 = ['uga', 'buga']
>>> for (a, b, c) in zip(lis1, lis2, lis3):
...     print(a, b, c)
1 a uga
2 b buga
```

Note that the length of the shortest list is the length of the zip. In the example above, **lis3** is of length 2 so we only got two printouts.

Chapter 10. In-place operations

Example. Let's write a function rev that reverses a list without going backwards. We will do it in two ways. 1. Returning a new list:

```
>>> def rev(lis):
...     newlis = [None]*len(lis)
...     for i,x in enumerate(lis):
...         newlis[len(lis)-1-i] = x
...     return newlis
...
>>> rev([1,2,3])
[3, 2, 1]
```

2. Changing the list in-place.

```
>>> def rev(lis):
...     for i in range(len(lis)//2):
...         r = len(lis)-1-i
...         lis[i], lis[r] = lis[r], lis[i]
...
>>> lis = [1,2,3]
>>> rev(lis)
>>> lis
[3, 2, 1]
```

Chapter 11. Sets

[set]

Python sets are unordered bags of objects. We can initialize them using braces **{}**. Just like sets in mathematics, repeated elements are ignored.

```
>>> myset = {'a','b','c','a','b'}
>>> myset
{'a', 'b', 'c'}
>>> len(myset)
3
```

A set can be initialized from any sequence,

```
>>> A = set([1,2])
>>> A
{1, 2}
>>> B = set(range(2))
>>> B
{0, 1}
>>> C = set('pizza')
>>> C
{'a', 'i', 'p', 'z'}
```

The string **'pizza'** is broken down to single letters because that's how Python strings behave when you iterate over them (e.g. using **for x in 'pizza'**). Sets can be iterated just like lists, but the order is not guaranteed.

```
>>> numbers = {5,-3,100}
>>> for num in numbers:
...     print(num)
...
5
100
-3
```

You can check for set equality using **A == B**, containment ($A \subseteq B$) using **A <= B** and proper containment ($A \subset B$) using **A < B**.

```
>>> {0,1,2} == {2,0,1,2}
True
>>> {1,2} <= {1,2}
True
>>> {1,2} < {1,2}
False
>>> {1,2} <= {1,99}
False
```

One of the most common operations on sets is to check if they contain an element.

```
>>> s = set(range(1000))
>>> 400 in s
True
>>> 2000 in s
False
```

The basic set operations (union, intersection, etc.) are summarized in Table 11.1.

Name	Operator	**A** Op **B**
Union	`|`	`{0,1,2}`
Intersection	`&`	`{1}`
Difference	`-`	`{2}`
Symmetric difference	`^`	`{0,2}`

Table 11.1: Set operations demonstrated on the sets `A={1,2}` and `B={0,1}`.

Python sets are mutable. You can use `set.add(element)` to add a single element and `set.remove(element)` to remove one.

```
>>> P = set('pizz')
>>> P
{'i', 'p', 'z'}
>>> P.add('a')
>>> P
{'a', 'i', 'p', 'z'}
>>> P.remove('p')
>>> P
{'a', 'i', 'z'}
```

If you try to remove an element that's not there you will get a `KeyError`

```
>>> P.remove('q')
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-268-5af8e465a877> in <module>
----> 1 P.remove('q')

KeyError: 'q'
```

All the binary operations in table 11.1 have update versions:

```
>>> A
{1, 2}
>>> B
{0, 1}
>>> A |= B
>>> A
{0, 1, 2}
```

Example. The popular game Wordle keeps track of the remaining letters that the user has not guessed. Let's emulate this in an example.

```
>>> letters = set(string.ascii_lowercase)
>>> print(letters)
{'g', 'x', 'o', 'v', 'u', 'm', 'q', 's', 'd', 'k', 'b', 'l', 'y', 'f', 'p',
    'h', 'e', 'i', 'r', 'j', 't', 'w', 'a', 'n', 'c', 'z'}
>>> letters -= set('cover')
>>> letters -= set('track')
>>> letters -= set('pings')
>>> letters
{'b', 'd', 'f', 'h', 'j', 'l', 'm', 'q', 'u', 'w', 'x', 'y', 'z'}
```

11.1 Set membership testing

Suppose we wish to correct a pizza order by removing all the awful toppings from it.

```
AWFUL_TOPPINGS = ['tuna', 'corn', 'egg', 'chicken', 'avocado']
def fix_pizza(toppings):
    return [x for x in toppings if x not in AWFUL_TOPPINGS]
```

We are using the list **AWFUL_TOPPINGS** just to test for membership. When Python runs **x not in AWFUL_TOPPINGS** it scans over the entire list. This can be very slow if the list of awful toppings is large! Sets are a data structure that is purpose-built for fast membership testing. So we can obtain a faster version of the above code just by setting **AWFUL_TOPPINGS = {'tuna', 'corn', 'egg', 'chicken', 'avocado'}**.

[timeit] This makes a huge difference for large lists. Let's test this with some IPython magic.

```
>>> L = list(range(1000000))
>>> %timeit -n100 500000 in L
2.34 ms ± 42.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

(your numbers will vary) The command **%timeit -n100** tells IPython to run the command that follows 100 times in a loop. This is done 7 times then the results are summarized. We learn that the command **500000 in L** takes about 4.91 milliseconds. This is a lot!

Since Python scans the list from beginning to end, we expect that the running time depends on the location of the element in the list. Let's see if that is the case.

```
>>> %timeit -n100 0 in L     # Searching for the first element
14.9 ns ± 3.48 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> %timeit -n100 999999 in L     # Searching for the last element
4.65 ms ± 54.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Finding the first element in this list takes about 15 *nanoseconds* (billionths of a second). Finding the last element takes 4.65 *milliseconds* (thousands of a second) - a huge number in computer speeds.

Using a set, we get fast lookups for all elements.

```
>>> S = set(L)
>>> %timeit -n100 0 in S
71.2 ns ± 26.9 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> %timeit -n100 999999 in S
64 ns ± 10.6 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

11.2 Set comprehensions

Set comprehensions are a simple syntax for building sets.

```
>>> {k**2 for k in range(-10,5) if (k%2)==1}
{1, 9, 25, 49, 81}
```

This is just like list comprehensions, but uses curly braces **{}** instead of **[]**.

11.3 Examples of using sets

Example. Given two lists ***list0, list1*** *find all the pairs of numbers (one from list0 and one from list1) that sum to 42*

```
>>> list0 = [1,6,17,3,-58,-5,20,31]
>>> list1 = [0, 54, 29, -17, 25, 100]
>>> for x in list0:
...     for y in list1:
```

```
...         if x+y == 42:
...             print(f'{x}+{y} == {x+y}')
...
```

Question. What's wrong with this solution?
Answer. It's OK for short lists, but slow when the lists are long. The nested loop performs **`len(list0)`** × **`len(list1)`** additions and comparisons, so if both lists have millions of elements it will run trillions of operations. Here's a much faster solution:

```
>>> setlist0 = set(list0)
>>> for y in list1:
...     if 42-y in setlist0:
...         print(f'{42-y}+{y} == 42')
...
```

The reason that this solution is faster is that **`42-y in setlist0`** takes a the check

Example. Given a list of names and birthdays, print all the birthdays and people that share a birthday Let us write some code that finds birthday collisions.

```
BIRTHDAYS = [('Danielle', (12,7)), ('Ben', (8,31)), ('Katya', (12,7)),
    ('Avigail', (6,15)), ('Joshua', (12,7))]
>>> bdays_seen = set()
>>> for (name, (month, day)) in BIRTHDAYS:
...     if (month, day) in bdays_seen:
...         print(f"Found collision: {name}'s birthday is the same as
    someone else's")
...     bdays_seen.add((month, day))
...
Found collision: Katya's birthday is the same as someone else's
Found collision: Joshua's birthday is the same as someone else's
```

The problem here is that we only see the collision on the second birthday. We can't tell who the first person is.

```
>>> bdays_seen = set()
>>> bdays_seen_twice = set()
>>> for (name, (month, day)) in BIRTHDAYS:
...     if (month, day) in bdays_seen:
...         bdays_seen_twice.add((month, day))
...     bdays_seen.add((month, day))
... for (name, (month, day)) in BIRTHDAYS:
...     if (month, day) in bdays_seen_twice:
...         print(f"{name}'s birthday appears more than once")
...
Danielle's birthday appears more than once
Katya's birthday appears more than once
Joshua's birthday appears more than once
```

This is quite cumbersome. What if we want to group the names by birthday? It's not easy to do using sets! The problem is that the **`set`** data structure only holds the birthdays but cannot store the names attached to the birthdays. In the next section we will learn about the **`dict`** data structure, that will make the solution of this problem easier.

Chapter 12. Dictionaries

12.1 Motivation

You are in charge of government statistics and your task is to analyze the distribution of the number of children per household. The input is a list of numbers **data**, where each elements is the number of kids of a particular household. For this you first compute a histogram:

```
>>> data = [0,1,0,3,2,3,5,0,2,7,4,2,0,3,1]
>>> hist = [0]*100
>>> for numkids in data:
...     hist[numkids] += 1
...
>>> hist
[4, 2, 3, 3, 1, 1, 0, 1]
>>> for (numkids, count) in enumerate(hist):
...     print(f'#kids={numkids} count={count}')
...
#kids=0 count=4
#kids=1 count=2
#kids=2 count=3
#kids=3 count=3
#kids=4 count=1
#kids=5 count=1
#kids=6 count=0
#kids=7 count=1
```

One funny thing about this code is that the length of the array **hist** depends on the maximum element of **data**. What if we want instead to compute the histogram of letters in a text string? Let's compute the letter histogram of the following quote by Donald Trump: *"I'm speaking with myself, number one, because I have a very good brain and I've said a lot of things."*

We need a way to assign a number to each letter. This is exactly what the **ord** function does.

[ord]

```
>>> s = "I'm speaking with myself, number one, because I have a very good
    brain and I've said a lot of things."
>>> ord(s[0]), ord(s[1]), ord(s[2])
(73, 39, 109)
```

Now we can initialize an array of counts, as before.

```
>>> hist = [0]*(max(map(ord, s))+1)
>>> for c in s:
...     hist[ord(c)] += 1
...
>>> print(hist)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 19, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 2, 0, 1, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    8, 3, 1, 3, 9, 2, 3, 3, 5, 0, 1, 2, 3, 6, 5, 1, 0, 3, 5, 3, 2, 3, 1, 0,
    2]
```

Now it is no longer clear how to to match the index of a count to the original character!

[chr] Luckily, Python has the **chr** which gives gives back the character given its number.

```
>>> ord('a')
97
>>> chr(_)
'a'
```

Now we can display our character histogram:

```
>>> for (i, count) in enumerate(hist):
...     if count > 0:
...         print(f'character: "{chr(i)}" count: {count}')
...
character: " " count: 19
character: "'" count: 2
character: "," count: 2
character: "." count: 1
character: "I" count: 3
character: "a" count: 8
character: "b" count: 3
character: "c" count: 1
character: "d" count: 3
character: "e" count: 9
character: "f" count: 2
character: "g" count: 3
character: "h" count: 3
character: "i" count: 5
character: "k" count: 1
character: "l" count: 2
character: "m" count: 3
character: "n" count: 6
character: "o" count: 5
character: "p" count: 1
character: "r" count: 3
character: "s" count: 5
character: "t" count: 3
character: "u" count: 2
character: "v" count: 3
character: "w" count: 1
character: "y" count: 2
```

(note that we only printed the characters with non-zero cells).

To summarize, there are two issues with our approach:

1. We need some way to match each character with an array index – and a way to remember or recover the character.

2. It can lead to very large **hist** lists. For example, if our text string included emoji, **ord('\N{grinning face}')** is 128512. Yikes.

What we really want here is a data type similar to a Python list, but one that can be indexed using characters.

12.2 The dict type

A Python dictionary, or **dict**, is a data structure that maps *keys* to *values*.:

```
>>> d = {'a': 5, 'b': 'Hello', 10: [1,2,3]}
{'a': 5, 'b': 'Hello', 10: [1, 2, 3]}
>>> d['a']
5
```

Here, **'a'** is a key and **5** is its value. Dictionaries can be modified:

```
>>> d['b'] = 'Goodbye' # Assign new value
>>> d['a'] += 1        # Modify value
>>> d['c'] = 123       # Set new key-value pair
>>> d
{'a': 6, 'b': 'Goodbye', 10: [1, 2, 3], 'c': 123}
```

In a Python dictionary, the values can be changed but keys can never be modified. To enforce this, Python requires that they be of immutable type (e.g. **string**, **int**, **float**, **bool**). In contrast, values in a dictionary can be anything. *Example. (histogram)* Compute a letters histogram using a dict.

```
>>> s = "I'm speaking with myself, number one, because I have a very good
    brain and I've said a lot of things."
>>> hist = {}
>>> for c in s:
...     hist[c] += 1
...
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-193-332787c69350> in <module>
      1 for c in s:
----> 2     hist[c] += 1
      3

KeyError: 'I'
```

Whoops! The problem here is that we are trying to do **hist['I'] += 1** but the dictionary **hist** does not have this key. We can check if a key exists in a dictionary using the **in** operator.

```
>>> for c in s:
...     if c not in hist:
...         hist[c] = 0
...     hist[c] += 1
...
>>> hist
{'I': 3,
 "'": 2,
 'm': 3,
 ' ': 19,
 's': 5,
 'p': 1,
 'e': 9,
 'a': 8,
 'k': 1,
 'i': 5,
 'n': 6,
 'g': 3,
 'w': 1,
 't': 3,
 'h': 3,
 'y': 2,
 'l': 2,
 'f': 2,
 ',': 2,
```

```
 'u': 2,
 'b': 3,
 'r': 3,
 'o': 5,
 'c': 1,
 'v': 3,
 'd': 3,
 '.': 1}
```

Example. (grouping) Given a list of words, group them by length.

We can solve this by building a dictionary that maps from a word length to a list of words of that length. To build the dictionary This means each value needs to hold a list, starting with the empty list and appending as we go along the words.

```
>>> for cheese in cheese_types:
...     k = len(cheese)
...     if k not in d:
...         d[k] = []
...     d[k].append(cheese)
...
>>> d
{10: ['mozzarella', 'gorgonzola'],
 8: ['parmesan', 'manchego'],
 5: ['gouda'],
 4: ['brie'],
 9: ['camambert', 'bulgarian'],
 7: ['cottage']}
```

Let's display the same dictionary, but sorted by length.

```
>>> sorted(d.items())
[(4, ['brie']),
 (5, ['gouda']),
 (7, ['cottage']),
 (8, ['parmesan', 'manchego']),
 (9, ['camambert', 'bulgarian']),
 (10, ['mozzarella', 'gorgonzola'])]
```

[tuple ordering]

Why does this sorting work? The items of a dictionary are tuples (length,words). When the **`sorted`** function compares the items of the list it uses the regular **`<`** logical comparison operator. When two tuples are compared via **`(a,b) < (c,d)`** the result is True if and only if either **`a < c`** or **`a==c`** and **`b<d`**.

12.3 Dict initialization

There are several ways to initialize a dictionary. We can start with an empty **`dict`** and add elements one-by-one:

```
>>> d = {}
>>> d{'firstname'] = 'John'
>>> d['firstname'] = 'John'
>>> d['lastname'] = 'Doe'
>>> d
{'firstname': 'John', 'lastname': 'Doe'}
```

Or we can initialize the dictionary with all its key-value pairs in place

```
>>> {'firstname': 'John', 'lastname': 'Doe'}
{'firstname': 'John', 'lastname': 'Doe'}
>>> dict([('firstname', 'John'), ('lastname', 'Doe')])
{'firstname': 'John', 'lastname': 'Doe'}
```

```
>>> dict(firstname='John', lastname='Doe')
{'firstname': 'John', 'lastname': 'Doe'}
```

The last style only works when the keys are strings. Of course, you can initialize a dictionary and then add or edit it.

```
>>> d = {'firstname': 'John', 'lastname': 'Doe'}
>>> d['lastname'] = 'Deer'
>>> d
{'firstname': 'John', 'lastname': 'Deer'}
```

Just like lists, we can use *dictionary comprehensions* to initialize a **dict**.

```
>>> words = 'Hi there first year students'.split(' ')
>>> words
['Hi', 'there', 'first', 'year', 'students']
>>> {word: len(word) for word in words if word.islower()}
{'there': 5, 'first': 5, 'year': 4, 'students': 8}
```

12.4 Dict iteration

We can get all the keys of a dictionary, all the values, or all the items (key-value pairs).

```
>>> d.keys()
dict_keys(['firstname', 'lastname'])
>>> d.values()
dict_values(['John', 'Deer'])
>>> d.items()
dict_items([('firstname', 'John'), ('lastname', 'Deer')])
```

Iterating over a dictionary does not give us the key-value pairs, but just the keys.

```
>>> for k in d:
...     print(k)
...
firstname
lastname
```

To get the key-value pairs we need to use **dict.items**

```
>>> for (k,v) in d.items():
...     print(k,v)
...
firstname John
lastname Deer
```

12.5 dict editing

One can merge dictionaries using the **.update()** method.

```
>>> d = {'firstname': 'John', 'lastname': 'Deer'}
>>> d.update({'middlename': 'F', 'lastname': 'Kennedy'})
>>> d
{'firstname': 'John', 'lastname': 'Kennedy', 'middlename': 'F'}
```

Whenever there's a conflict, the last dictionary wins.

Dictionary elements can be deleted with **del**

```
>>> d = {'firstname': 'John', 'lastname': 'Kennedy', 'middlename': 'F'}
>>> del d['middlename']
>>> d
{'firstname': 'John', 'lastname': 'Kennedy'}
```

12.6 Birthday collisions with dict

Example. Given a list of names and birthdays, print all the birthdays and people that share a birthday.

```
BIRTHDAYS = [('Danielle', (12,7)), ('Ben', (8,31)), ('Katya', (12,7)),
    ('Avigail', (6,15)), ('Joshua', (12,7))]
for (name, bday) in BIRTHDAYS:
    if bday not in d:
        d[bday] = []
    d[bday].append(name)
```

12.7 Dict example: censorship

```
CENSORSHIP = {'secret': 'XXXXX', 'pizza': '!!!'}

def censor(word):
    if word in CENSORSHIP:
        return CENSORSHIP[word]
    else:
        return word

def censor_message(message):
    words = message.split(' ')
    censored_words = map(censor, words)
    return ' '.join(censored_words)

>>> message = 'This is a very secret message about pizza. Shhhhh'
>>> censor_message(message)
'This is a very XXXXX message about pizza. Shhhhh'
```

12.8 `defaultdict`

Example. (back to histograms) In the particular case of a histogram, there is a more elegant solution than the one we showed above. The **`collections.defaultdict`** data type behaves like a dict where all values are already initialized.

```
>>> import collections
>>> hist = collections.defaultdict(lambda:0)
>>> for c in s:
...     hist[c] += 1
...
>>> hist
defaultdict(<function __main__.<lambda>()>,
            {'I': 3,
             "'": 2,
             'm': 3,
             ' ': 19,
             .
             .
             .
             'd': 3,
             '.': 1})
```

The **`lambda:0`** in the call to **`defaultdict`** is the **`default_factory`** argument. Every time **`hist[k]`** is called, the defaultdict checks if the key **`k`** already exists in the dictionary. If not then it performs **`hist[k] = default_dict()`**.

Example. (back to grouping) Given a list of words, group them by length.

```
>>> d = collections.defaultdict(lambda:[])
>>> for cheese in cheese_types:
...     d[len(cheese)].append(cheese)
...
>>> d
defaultdict(<function __main__.<lambda>()>,
            {10: ['mozzarella', 'gorgonzola'],
             8: ['parmesan', 'manchego'],
             5: ['gouda'],
             4: ['brie'],
             9: ['camambert', 'bulgarian'],
             7: ['cottage']})
```

Chapter 13. Python scripts

So far we have seen two ways to run Python code:

- Running code in the Python interpreter.
- Importing code as a Python module.

Another way is to run a Python program from the command-line (such programs are often called "Python scripts"). Open up the following program as **hithere.py** in the local directory **code**

```
print('Hi there!')
```

Try to run it from the command-line by typing **python code/hithere.py** in the command line (note to Windows users: make sure you run this from the Anaconda prompt). Command-line programs often take arguments that you can access using **sys.args** in the **sys** module. Let's see this in action:

[sys.argv]

```
import sys
print('Hi there!')
print(sys.argv)
```

If we now type **code/python hithere.py bob taylor** we get the following:

```
> python hithere.py bob taylor
Hi there!
['code/hithere.py', 'bob', 'taylor']
```

We see that **sys.argv** is a list of all the command-line arguments as strings, preceded by the path of the prorgram.
Question. How would you change **hithere.py** to say "Hi there **name**!" to each name given as a command-line argument? *Answer:*

```
import sys

def greet(names):
    for name in names:
        print(f'Hi there {name}!')

print(f'Greeting {len(sys.argv)-1} names given as command-line arguments:')
greet(sys.argv[1:])
```

Let's try to run this:

```
> python ../code/hithere.py Bob Taylor
Greeting 2 names given as command-line arguments:
Hi there Bob!
Hi there Taylor!
```

To greet names that contain spaces you will have to surround them with "":

```
python ../code/hithere.py Bob "Taylor Swift"
Greeting 2 names given as command-line arguments:
Hi there Bob!
Hi there Taylor Swift!
```

13.1 importing Python scripts

Every **.py** file can be imported as a module. Sometimes we will also want to import Python scripts to use some of their functionality, to test them, etc. Let's see what happens when we import our file:

```
>>> import hithere
Greeting 0 names given as command-line arguments:
```

Argh. We just wanted to import the function **greet** but since importing runs the file, we also got an unwanted print. Python defines a special variable **__main__** that contains the name of the current module. Let's print it and see what happens:

```
import sys

def greet(names):
    for name in names:
        print(f'Hi there {name}!')

print(f'{__name__=}')
print(f'Greeting {len(sys.argv)-1} names given as command-line arguments:')
greet(sys.argv[1:])
```

[f'varname=]

Note that **f'varname='** is a nice shorthand for **f'varname=varname**. Running this from the command-line gives:

```
python ../code/hithere.py abc XYZ
__name__='__main__'
Greeting 2 names given as command-line arguments:
Hi there abc!
Hi there XYZ!
```

However, if we import **hithere.py** we get the following output:

```
    import hithere
    __name__='hithere'
    Greeting 0 names given as command-line arguments:
```

To summarize, **__name__** is a special variable.

- When importing a module, **__name__** is equal to the module name.
- When running the script from the command-line, **__name__** is equal to **'__main__'**.

Python scripts are typically organized as follows:

```
def greet(names):
    for name in names:
        print(f'Hi there {name}!')

def main():
    import sys # We only need to import sys when running as a Python script
    print(f'Greeting {len(sys.argv)-1} names given as command-line
    arguments:')
    greet(sys.argv[1:])

if __name__ == '__main__':
    main()
```

This way we can either run the script from the command-line or import it into Python to use the function **greet**:

```
>>> import hithere
>>> hithere.greet(['falafel','bourekas'])
```

Chapter 14. Reading and writing text files

14.1 Reading text files

In this example we will use the Iris dataset that contains measurements of flowers from several different species. It is available here:
https://archive.ics.uci.edu/ml/datasets/iris
To open a text file for reading use the **open** command. This command takes a filename and returns a *file object* that we can read from:

```
>>> f = open('iris.data')
>>> text = f.read()
>>> text
'5.1,3.5,1.4,0.2,Iris-setosa\n4.9,3.0,1.4,0.2,Iris-setosa\n4.7,3.2,1.3,0.2,
Iris-setosa\n4.6,3.1,1.5,0.2,Iris-setosa\n5.0,3.6,1.4,0.2,Iris-setosa\n5.4,
3.9,1.7,0.4,Iris-setosa\n4.6,3.4,1.4,0.3,Iris-setosa\n5.0,3.4,1.5,0.2,
    .
    .
    .
6.2,3.4,5.4,2.3,Iris-virginica\n5.9,3.0,5.1,1.8,Iris-virginica\n\n'
```

We got all the text lines and newline characters **'\n'** jumbled together in one big string. We could use **data.split('\n')**, but instead we'll demonstrate the built-in **.readlines()** method.

```
>>> f = open('iris.data')
>>> f.readlines()
['5.1,3.5,1.4,0.2,Iris-setosa\n',
 '4.9,3.0,1.4,0.2,Iris-setosa\n',
 '4.7,3.2,1.3,0.2,Iris-setosa\n',
 .
 .
 .
 '5.9,3.0,5.1,1.8,Iris-virginica\n',
 '\n']
```

Note that the **'\n'** is still there at the end of every line. Text files created on Windows often have newlines made of two characters: **'\r\n'** (an ancient relic of the past). The **.readlines()** method can handle those just as well. Let's try reading more:

```
>>> f.read()
''
>>> f.readlines()
[]
```

What happened? The file object **f** holds the current position in the file, which advances every time we read. After reading the entire file, the file is exhausted. We can move the position back to the start of the file using **f.seek(0)**, but it is more common to just reopen the file.

```
>>> f = open('iris.data')
>>> f.readline()
'5.1,3.5,1.4,0.2,Iris-setosa\n'
>>> f.readline()
'4.9,3.0,1.4,0.2,Iris-setosa\n'
>>> for line in f:
...     print(line)
...
4.7,3.2,1.3,0.2,Iris-setosa

4.6,3.1,1.5,0.2,Iris-setosa

5.0,3.6,1.4,0.2,Iris-setosa
.
.
.
```

Question. Why is there an empty line between every printed line?
Answer. Because, the command **`print(line)`** involves two newline characters, one at the end of **`line`** and one that is added by the **`print`** command. You may recall that **`print`** takes a keyword argument **`end`** with a default value of **`'\n'`** To fix this, we can ask the **`print`** command to not add a newline at the end of the print like this:

```
>>> for line in open('iris.data'):
...     print(line, end='')
...
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
.
.
.
```

14.1.1 Digging into the Iris data set

Example. How many different species are in the Iris dataset? First, let's read all the lines. Then take a single line and try to extract the species.

```
>>> f = open('iris.data')
>>> lines = f.readlines()
>>> lines[0].split(',')
['5.1', '3.5', '1.4', '0.2', 'Iris-setosa\n']
>>> lines[0].split(',')[-1]
'Iris-setosa\n'
```

To remove the trailing newline, we can use **`.strip()`**. We can extract all the Iris species using a list comprehension.

```
>>> [line.split(',')[-1].strip() for line in lines]
['Iris-setosa',
 'Iris-setosa',
 .
 .
 .
 'Iris-virginica',
 'Iris-virginica',
 '']
```

The empty element at the end of the list comes from the traling empty line at the end of the file. To get the number of different species we can use a set comprehension.

```
>>> {line.split(',')[-1].strip() for line in lines}
{'', 'Iris-setosa', 'Iris-versicolor', 'Iris-virginica'}
```

We see that there are just 3 species and one bogus "species". Had we just computed the size of the set using **`len()`** we would have reached the wrong result. It is always important to look at the data carefully!

Example. What is the range of the first column values? We'll start by reading the lines and splitting by commas:

```
>>> lines = open('iris.data').readlines()
>>> [line.split(',') for line in lines[:-1]]
[['5.1', '3.5', '1.4', '0.2', 'Iris-setosa\n'],
 ['4.9', '3.0', '1.4', '0.2', 'Iris-setosa\n'],
 .
 .
 .
 ['5.9', '3.0', '5.1', '1.8', 'Iris-virginica\n']]
```

The first column is composed of all the first elements. Let's convert them from strings to numbers and build a new list:

```
>>> firstcol = [float(line.split(',')[0]) for line in lines[:-1]]
>>> firstcol
[5.1,
 4.9,
 4.7,
 .
 .
 .
 5.9]
```

Finally, we can obtain the range of the first column by taking a minimum and maximum:

```
>>> min(firstcol), max(firstcol)
(4.3, 7.9)
```

14.1.2 Writing a Grep utility

Grep is a famous command-line utility for printing lines with a given keyword in an input text file. Programmers often use it to locate TODO comments, look for specific dates in a log file, etc. In this section we will develop a simple grep script that can be used like this to find all occurences of the word "tenth" in the book Moby Dick:

```
> python code/grep.py tenth moby-dick.txt
618:  "A tenth branch of the king's ordinary revenue, said to be grounded
10778:they have of their whaling scenes. With not one tenth of England's
15645:nominally supplying the tenth branch of the crown's ordinary revenue. I
17525:that part, the remaining ribs diminished, till the tenth and last only
```

Printing all the lines that match a certain keyword is easy:

```
for line in f:
    if searchword in line:
        print(line, end='')
```

This shows us that the word "tenth" appears 4 times in the book Moby We will implement a simple

14.2 Writing text files

To open a text file for writing we use **`open(filename, 'wt')`**. The second parameter is the *mode* in which the file is to be opened. The flag **`'wt'`** means open for **w**riting in **t**ext mode. Text mode is the default so we don't have to specify it (but it is better to be explicit).

[write]

```
>>> f = open('testing.txt','wt')
>>> f.write('Hello')
5
>>> f.write(' world!')
7
```

If you now try to open the file you will probably see that it's empty! For efficiency reasons calls to **`.write()`** do not immediately write to the file. They fill a buffer in memory and occassionally write it out to the disk. There are two ways to force the data to be written:

- (less common) Call **`.flush()`** to write all the data that's currently buffered.
- (more common) Call **`.close()`** to flush the data and close the file. In this case, you can no longer write to the file.

[close]

Let's try closing the file and then reading what we wrote.

```
>>> f.close()
>>> open('testing.txt').read()
'Hello world!'
```

Here's a typical pattern for writing results to a file:

```
def write_results(filename, results):
    f = open(filename, 'wt')
    for x in results:
        line = process_result(x)
        f.write(line)
    f.close()
```

Just to have a simple concrete example, lets have **`results`** be a list of numbers and the processing function be **`math.sqrt`**.

```
from math import sqrt
def write_results(filename, results):
    f = open(filename, 'wt')
    for x in results:
        f.write(f'{str(sqrt(x))}\n')
    f.close()
```

Try to call this function like this: **`write_results('tmp.txt', [1,2,3])`**. A file named tmp.txt should be created in the current directory with the three square roots, one on each line.

14.2.1 Errors while writing files

Question. What happens if an error occurs while writing to a file? For example, in **`write_results`** if **`results`** contains a negative number. *Answer.* The call to **`sqrt`** will trigger an exception and the function **`write_results`** will terminate without ever calling **`f.close()`**. This means that some (or all) of what we wrote to the file is never flushed, so we will not see it in the file. This is usually considered bad behavior and can

make debugging more difficult. How do we make sure that the file contains all the text that was written right until the moment that the exception occured? There are a few different ways:

1. Calling **`f.flush()`** after every write command. This is very inefficient. We could also forget to flush every time.

2. Adding code to handle the exceptions and call **`f.flush()`** or **`f.close()`** whenever any exception occurs. We will learn how to do this in Section 20. This solution is the most flexible and allows us to add additional error-handling such as calling **`f.write('Failed due to exception!\n')`** before closing the file. However, it is cumbersome.

3. **(Recommended)** Using a **`with`** block:

[with]

```
def write_results(filename, results):
    with open(filename, 'wt') as f:
        for x in results:
            f.write(f'{str(sqrt(x))}\n')
```

The **`with`** block guarantees that **`f.close()`** is called at the end of the block whether or not an exception occured.

14.2.2 Appending to files

Files can also be opened in **a**ppend mode:

```
>>> f = open('tmp.txt', 'w')
>>> f.write('Blah blah.\n')
11
>>> f.close()
>>> print(open('tmp.txt').read())
Blah blah.

>>> f = open('tmp.txt', 'a')
>>> f.write('Adding another line...\n')
23
>>> f.close()
>>> print(open('tmp.txt').read())
Blah blah.
Adding another line...
```

Example. Let's write a small script for tracking working hours.

```
import sys
from datetime import datetime

WORKHOURSLOG = 'timelog.txt'

def trackhours(task):
    with open(WORKHOURSLOG, 'at') as f:
        starttime = datetime.now().isoformat(sep=' ', timespec='seconds')
        input('Hit enter when done')
        endtime = datetime.now().isoformat(sep=' ', timespec='seconds')
        f.write(f'Start: {starttime} | End: {endtime} | Task: {task}\n')

def main():
    if len(sys.argv) != 2:
        print(f'Usage: python {sys.argv[0]} <activity>')
```

```
        return
    trackhours(sys.argv[1])

if __name__=='__main__':
    main()
```

Chapter 15. Saving and loading data

Suppose we have have a variable with some data that we wish to save to a file.

```
>>> data = [[1, 2, 3, 4], [5, 'six']]
>>> with open('tmp.txt','wt') as f:
...     f.write(data)
...
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-0ddcdb17df3b> in <module>
      1 with open('tmp.txt','wt') as f:
----> 2     f.write(data)
      3

TypeError: write() argument must be str, not list
```

This doesn't work because you can only write strings to a text file. The **repr** function converts Python data objects to a string that can later be evaluated in Python.

[repr]

```
>>> repr(data)
"[[1, 2, 3, 4], [5, 'six']]"

>>> with open('tmp.txt','wt') as f:
...     f.write(repr(data))
```

We can then read the string and copy&paste it in the interpreter.

```
>>> reprdata = open('tmp.txt').read()
>>> reprdata
"[[1, 2, 3, 4], [5, 'six']]"
>>> loaded_data = [[1, 2, 3, 4], [5, 'six']]
```

The programmatic way to do this is to use **eval**.

```
>>> eval(reprdata)
[[1, 2, 3, 4], [5, 'six']]
```

The function **eval** uses the Python interpreter to run the code given in the input string.

[eval]

```
>>> import math
>>> eval('math.sin(1)')
0.8414709848078965
```

This approach for saving/loading using **repr**/**eval** has one nice advantage: the files created are human-readable. It has several major disadvantages:

1. It is dangerous! Calling **eval** is not always safe! A malicious intruder can replace your file with a command that deletes the entire hard drive!
2. It creates large files.
3. It is slow.

There are better ways to save and load data. Python can interface with lots of standard file formats (CSV, JSON, XML, YAML, etc.). You can also make up your own formats for reading and writing data.

15.1 repr vs str

You may have noticed that **repr** is very similar to **str**.

```
>>> lis = [1,2,3]
>>> s = 'abc'
>>> str(lis)
'[1, 2, 3]'
>>> str(s)
'abc'
>>> repr(lis)
'[1, 2, 3]'
>>> reps(s)
>>> repr(s)
"'abc'"
```

The difference is that **str** is meant to look nice whereas **repr** is supposed to give a valid Python expression that can be entered into the Python interpreter. Here's an example of the difference using a string object:

```
>>> s = 'abc'
>>> str(s)
'abc'
>>> repr(s)
"'abc'"
```

And here's another example using datetime.

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> str(now)
'2022-12-06 14:26:04.624334'
>>> repr(now)
'datetime.datetime(2022, 12, 6, 14, 26, 4, 624334)'
>>> t = datetime.datetime(2022, 12, 6, 14, 26, 4, 624334)
>>> print(t)
```

15.2 pickle

[pickle]

The built-in **pickle** module is the standard module for saving and loading Python objects to files. It uses a special binary format.

```
>>> import datetime
>>> import pickle
>>> pickle.dumps(datetime.datetime.now())
b'\x80\x04\x95*\x00\x00\x00\x00\x00\x00\x00\x8c\x08datetime\x94\x8c\x08date
time\x94\x93\x94C\n\x07\xe6\x0c\x06\x0e\x1a\x04\t\x86\xce\x94\x85\x94R\x94.'
>>> pickle.loads(_)
datetime.datetime(2022, 12, 6, 14, 26, 4, 624334)
```

Note the preceding **b** before the output of **pickle.dump**. This is a **bytes** object that represents a sequence of bytes (each byte is 8 bits). Regular strings represent sequences of unicode characters – they don't have a single binary encoding.

Pickle has the **dump** and **load** methods that work directly with Python file objects.

```
>>> with open('time.pickle', 'wb') as f:
...     pickle.dump(now, f)
...
>>> pickle.load(open('time.pickle', 'rb'))
datetime.datetime(2022, 12, 6, 14, 26, 4, 624334)
```

Functions can be pickled too!

```
>>> pickle.load(open('time.pickle', 'rb'))
<function __main__.func(x)>
>>> with open('function.pickle', 'wb') as f:
...     pickle.dump(func, f)
...
>>> pickle.load(open('function.pickle', 'rb'))
<function __main__.func(x)>
```

Chapter 16. File operations

```
>>> import os
>>> os.path.exists('time.pickle')
True
>>> os.path.exists('thump.pickle')
False
```

Let's create a directory "blah" under the current directory, then check if it's a directory:

```
>>> !mkdir blah
>>> os.path.isfile('blah')
False
>>> os.path.isdir('blah')
True
```

Rename:

```
>>> os.rename('time.pickle', 'hmpf')
```

Remove a file:

```
>>> os.path.exists('hmpf')
True
>>> os.remove('hmpf')
>>> os.path.exists('hmpf')
False
```

Let's make and then delete a directory:

```
>>> os.mkdir('d')
>>> os.rmdir('d')
```

We can get the current directory using **`os.getcwd()`** (short for "get current working directory"):

```
>>> os.getcwd()
'/Users/mosco/Dropbox/teaching/2022a introduction to computers for
   statisticians/lecture-notes/class 6 - file operations'
```

The function **`os.listdir(dirname)`** returns the list of files and directories inside the directory **`dirname`**. To get the list of files in the current directory we can use **`os.listdir()`** or **`os.listdir('.')`**, since "." always refers to the current directory.

```
>>> os.listdir()
['tmp.txt',
 'blah',
 'tmp.pickle',
 'function.pickle']
```

To get the full path of one of these file names, we can use **`os.path.abspath`**.

```
>>> os.path.abspath('tmp.txt')
'/Users/mosco/Dropbox/teaching/2022a introduction to computers for
   statisticians/lecture-notes/class 6 - files, error handling/tmp.txt'
```

The function **abspath** just sticks the path of the current working directory and the input filename together. What if we list a directory other than the current working directory? Let's try placing a file **blahfile.txt** inside the directory **blah** and then using **os.listdir**:

```
>>> files = os.listdir('blah')
>>> files
['blahfile.txt']
>>> os.path.abspath(files[0])
'/Users/mosco/Dropbox/teaching/2022a introduction to computers for
    statisticians/lecture-notes/class 6 - file operations/blahfile.txt'
>>> os.path.join(os.getcwd(), 'blah', files[0])
'/Users/mosco/Dropbox/teaching/2022a introduction to computers for
    statisticians/lecture-notes/class 6 - file operations/blah/blahfile.txt'
```

Whoops, this is the wrong path for **blahfile.txt**. The function **abspath** doesn't know that **blahfile.txt** is inside a subdirectory. We need to manually join the parts of the path using **os.path.join**.

[os.path.join]

```
>>> os.path.join(os.getcwd(), 'blah', files[0])
'/Users/mosco/Dropbox/teaching/2022a introduction to computers for
    statisticians/lecture-notes/class 6 - file operations/blah/blahfile.txt'
```

Note: On Unix systems the slash symbol **'/'** is used for file paths whereas Windows uses backslash symbols **'\'**. The function **os.path.join** uses the standard type of slash for the operating system. However, you can use slashes in all of Python's file manipulation functions even on a Windows machine. e.g. **'C:/Users/MyUsername/'**...

Part II

Intermediate Python

Chapter 17. Comprehensions and functional programming

17.1 List comprehensions

A lot of what we did so far has the general pattern of building a list from a list, with optional filtering and function application. This pattern is so common that Python has a magical syntax to support it: *list comprehensions*. They combine the power of **`map`**, **`filter`** and more.

Example. Using a list comprehension to cube numbers in a list.

```
>>> lis = [5, 0, 10, -1, 7]
>>> [x**3 for x in lis]
[125, 0, 1000, -1, 343]
```

Example. Using a list comprehension to filter positive numbers.

```
>>> [x for x in lis if x > 0]
[5, 10, 7]
```

Example. Using a list comprehension to cube only the positive numbers.

```
>>> [x**3 for x in lis if x > 0]
[125, 1000, 343]
```

Nested loops are also supported by list comprehensions.

```
>>> [(x,y) for x in lis for y in ['a','b']]
[(5, 'a'),
 (5, 'b'),
 (0, 'a'),
 (0, 'b'),
 (10, 'a'),
 (10, 'b'),
 (-1, 'a'),
 (-1, 'b'),
 (7, 'a'),
 (7, 'b')]
```

Example. Given a list of numbers, build a new list that contains all of the positive numbers and their negation. Here's the boring multi-line solution:

```
>>> out = []
>>> for x in lis:
...     if x > 0:
...         out.append(-x)
...         out.append(x)
...
>>> out
[-5, 5, -10, 10, -7, 7]
```

and here's a cool one-line list comprehension that gives the same result:

```
>>> [x*y for x in lis for y in [-1,+1] if x > 0]
[-5, 5, -10, 10, -7, 7]
```

Example. Multiplication table using list comprehensions:

```
>>> [[i*j for j in range(1,n+1)] for i in range(1,n+1)]
[[1, 2, 3, 4, 5],
 [2, 4, 6, 8, 10],
 [3, 6, 9, 12, 15],
 [4, 8, 12, 16, 20],
 [5, 10, 15, 20, 25]]
```

This is a list comprehension of the form **`[SOMETHING(i) for i in range(1,n+1)]`** where **`SOMETHING(i)`** is the list comprehension **`[i*j for j in range(1,n+1)]`**.

17.2 Set and dict comprehensions

Set-comprehensions are basically the same as list comprehensions but build a set:

```
>>> lis = [1,-5,3,5,6,100]
>>> {x**2 for x in lis}
{1, 9, 25, 36, 10000}
```

17.3 Functional programming

So far we have seen how to define and call functions. In the Python language, functions can be manipulated just like numbers or strings. Let's see some examples.

Functions can be assigned to variables:

```
>>> def f1():
...     print(1)
...
>>> type(f1)
function
>>> def f2():
...     print(2)
...
>>> f = f2
>>> f()
2
```

Note: to assign a function to a variable, we use the function name without parentheses!

```
def square(x):
    return x**2
def cube(x):
    return x**3
funclist = [square,cube]
[f(7) for f in funclist)]
```

```
>>> def printer(value):
...     print(value)
...
>>> def calltwice(f, x):
...     f(x)
...     f(x)
...
>>> calltwice(printer, 42)
42
42
```

Example. Writing a generic groupby functions

```
>>> from collections import defaultdict
>>> CHEESES = ['gouda', 'cascaval', 'parmesan', 'brie', 'bulgarian',
    'roumbiet', 'gorgonzola', 'cottage', 'camambert']
>>> def groupby(seq, keyfunc):
...     d = defaultdict(lambda:[])
...     for x in seq:
...         d[keyfunc(x)].append(x)
...     return d
...
>>> groupby(CHEESES, len)
defaultdict(<function __main__.groupby.<locals>.<lambda>()>,
            {5: ['gouda'],
             8: ['cascaval', 'parmesan', 'roumbiet'],
             4: ['brie'],
             9: ['bulgarian', 'camambert'],
             10: ['gorgonzola'],
             7: ['cottage']})
```

Functions can build and return new functions.

Example. In mathematics, the function composition $f \circ g$ is defined by $(f \circ g)(x) = f(g(x))$. Let's try to implement this concept in Python:

```
def compose(f,g):
    def fg(x):
        return f(g(x))
    return fg
```

Every time this function is called, it *defines* a new function **`fg`**. This new function **`fg`** takes an input **`x`** and calculates **`f(g(x))`**. Importantly, **`compose`** does not call the function **`fg`**. It merely defines (and returns) it. Let's try to use it:

```
>>> def incrementer(x):
...     return x+1
...
>>> def twicer(x):
...     return 2*x
...
>>> f = compose(incrementer, twicer)
>>> f(10)
21
>>> f = compose(twicer, incrementer)
>>> f(10)
22
```

The style of working with functions that act on other functions is called *functional programming*. It can be used to write very elegant code in many cases so it is worthwhile to get comfortable with it.

17.4 map

Functions can be arguments to other functions: *Example.* (map) A common task is to apply a function to each element of a list, producing a new list. An example of this is the **cuber** function that cubes every number.

```
>>> def cuber(lis):
...     cubed = []
...     for x in lis:
...         cubed.append(x**3)
...     return cubed
...
>>> cuber([1,2,3])
[1, 8, 27]
```

We can code this general pattern in a function **apply_to_all** that applies an input function **f** to every element of an input list.

```
>>> def apply_to_all(f, lis):
...     f_lis = []
...     for x in lis:
...         f_lis.append(f(x))
...     return f_lis
```

We can now cube elements very easily.

```
>>> def cube(x):
...     return x**3
...
>>> apply_to_all(cube, [1,2,3])
[1, 8, 27]
```

We can just as easily compute the absolute value of every number in a list, using the built-in **abs** function.

[abs]

```
>>> apply_to_all(abs, [5, -8, 1, -100])
[5, 8, 1, 100]
```

Python already has a function function called **map** that applies a function to every value in a sequence. However, it returns an iterator instead of a list (just like **range**). This is done for reasons of efficieny – this way, we don't have to keep all the values in memory.

[map]

```
>>> map(cube, [1,2,3])
<map at 0x7fb1d91ebca0>
>>> for value in map(cube, [1,2,3]):
...     print(value)
...
1
8
27
```

As with every iterator, we can always use **list()** to construct an explicit list.

```
>>> list(map(cube, [1,2,3]))
[1, 8, 27]
```

17.5 filter

The built-in function **filter** is used to get all the elements that satisfy a condition, where the condition is given by a function that returns **True** or **False** (such functions are called *logical predicates*).

[filter]

Example. Given a list of pizza toppings, we will produce a new list that only includes the toppings that are not awful.

```
>>> def is_not_awful(topping):
...     return topping not in ['tuna', 'chicken', 'egg', 'corn', 'avocado']
...
>>> list(filter(is_not_awful, ['tuna', 'tomato', 'corn', 'olives']))
['tomato', 'olives']
```

17.6 reduce

Another common pattern is to reduce a list to a single element by repeatedly applying a single binary (two-input) operation. For example, when we evaluate the sum **a+b+c+d** what is actually happening is 3 additions of two numbers: **((a+b)+c)+d**. Similarly, the maximum **max(a,b,c,d)** can be rewritten as **max(max(max(a,b),c),d)**. Do you see the pattern? This is what the **reduce** function in the **functools** module does. Given a list $[x_1, x_2, \ldots, x_n]$ and a function f that takes two arguments, it computes

$$\begin{aligned} R_2 &= f(x_1, x_2) \\ R_3 &= f(r_2, x_3) \\ &\vdots \\ R_n &= f(r_{n-1}, x_n) \end{aligned}$$

[reduce] *Example.* Let's use reduce to compute the maximum of a sequence.

```
>>> import functools
>>> functools.reduce(max, [2,7,5,4,9])
9
```

Reduce isn't really necessary here since **max** can take a list as input.

[operator] *Example.* Let's use **reduce** to compute the product of a list of numbers.

```
>>> import operator
>>>  functools.reduce(operator.mul, [5,7,10])
350
```

We didn't need to define a function **mul** that computes the product of its two inputs because the **operator** module conveniently has all the basic mathematical operations.

17.7 replace_if

Example. Given a list of numbers named numbers, replace every occurrence of 666 in the list numbers with the string GO AWAY SATAN!.
Question. How would you implement this task in the functional programming style?
Answer. Consider the general pattern of replacing every element that satisfies a condition with some fixed value. We will call this operation **replace_if**.

```
>>> def replace_if(seq, cond, replace_val):
...     out = []
...     for x in seq:
...         if cond(x):
...             out.append(replace_val)
...         else:
...             out.append(x)
...     return out
```

We can now use this function to solve our de-satanization task.

```
>>> def is_satanic(num):
...     return num == 666
...
>>> replace_if([2,4,6,666,667], is_satanic, 'GO AWAY SATAN!')
[2, 4, 6, 'GO AWAY SATAN!', 667]
```

7 Boom example:

```
def replace_multi_if(seq, conditions, replacewith):
    out = []
    for x in seq:
        for cond in conditions:
            replaced = False
            if cond(x):
                out.append(replacewith)
                replaced = True
                break
        if not replaced:
            out.append(x)
    return out

def divisible_by_7(num):
    return num % 7 == 0

def contains_7_digit(num):
    return '7' in str(num)
```

17.8 Lambda functions

In the functional programming style you often need to write tiny helper functions (such as **cube** and **is_satanic** in the previous sections). The *lambda* statement lets you define an unnamed function as follows:

```
>>> apply_to_all(lambda x: x**3, [1,2,3])
[1, 8, 27]
```

This is similar to defining a function and then deleting it,

```
>>> def my_function(x):
...     return x**3
...
>>> apply_to_all(my_function, [1,2,3])
[1, 8, 27]
>>> del my_function
```

Lambda functions can take more than one variable. Let's compute the product of a list of numbers using lambda:

```
>>> functools.reduce(lambda x,y: x*y, [5,7,10])
350
```

17.9 When to use the functional programming style

Since every problem can be viewed as a special case of a more general problem, it is tempting to try to *generalize all the things* and write everything in the functional programming style. However, this is often not necessary. In the example above, writing a more specific function **replace(seq, v1, v2)** that replaces occurrences of **v1** with **v2** is just as good and probably more readable.

So, when should we try to generalize our problems and solve the general problem in a functional style? This is a matter of taste, but here are two rules of thumb:

1. *Don't repeat yourself!* If you find yourself reusing the same pattern 2-3 times, consider rewriting the code so that the pattern appears only once.

2. *Can you name it?* If you can find a good name for a repeating pattern such as **`replace_if`**, that is a good hint that it should be coded into a function.

Chapter 18. Object-oriented programming

18.1 Cookie jars using dictionaries

Suppose we want to keep track of multiple cookie jars with multiple owners in a shared space. Each cookie jar has an owner, a non-negative number of cookies, and a maximum capacity. One way to represent a cookie jar is using a dictionary. Let's write an initialization function and two functions for adding cookies and eating a cookie.

```
def cookiejar_init(owner, capacity):
    return {'owner': owner, 'capacity': capacity, 'n_cookies': 0}

def cookiejar_add(jar, n):
    assert jar['n_cookies']+n <= jar['capacity'], 'Too many cookies
    \N{drooling face}'
    jar['n_cookies'] += n

def cookiejar_eat_cookie(jar):
    assert jar['n_cookies'] >= 0, 'NO MORE COOKIES!!!'
    jar['n_cookies'] -= 1
    print('Om Nom Nom Nom')
```

Now we can play with cookie jars:

```
>>> jar = cookiejar_init('Cookie monster', 7)
>>> jar
{'owner': 'Cookie monster', 'capacity': 7, 'n_cookies': 0}
>>> cookiejar_add(jar, 3)
>>> cookiejar_add(jar, 10)
>>> jar
{'owner': 'Cookie monster', 'capacity': 7, 'n_cookies': 3}
>>> cookiejar_eat_cookie(jar)
>>> jar
{'owner': 'Cookie monster', 'capacity': 7, 'n_cookies': 2}
```

It would be nice to have a dedicated print function:

```
def cookiejar_print(jar):
    print(f"=== {jar['owner']}'s cookie jar ===")
    print('\N{cookie}'*jar['n_cookies'])
```

In the next subsection we'll see how Python's object-oriented facilities let us implement the above in a nicer way.

18.2 Cookie jars using classes

We will reimplement the cookie jar using a Python *class*. This class will define the behavior of cookie-jar objects: how they are initialized, what data do they keep, and what are the available operations that they come equipped with.

[class]

```
class CookieJar:
    def __init__(self, owner, capacity):
        self.owner = owner
        self.capacity = capacity
        self.n_cookies = 0

    def add(self, n):
        assert self.n_cookies+n <= self.capacity, 'Too many cookies
    \N{drooling face}'
        self.n_cookies += n

    def eat_cookie(self):
        assert self.n_cookies >= 0, 'NO MORE COOKIES!!! \N{crying face}'
        self.n_cookies -= 1
        print('Om Nom Nom Nom')

    def __str__(self):
        heading = f"=== {self.owner}'s cookie jar ==="
        cookies = '[' + '\N{cookie}'*self.n_cookies + '
    '*(self.capacity-self.n_cookies) + ']'
        return '\n'.join([heading, cookies])
```

[members] The data fields of an object are called *members*. In this example those are **owner, capacity, n_cookies**. The functions that are part of the class definition are called [methods] *methods*. Their first argument always points to the current object (similar to the *this* keyword in C++ and Java). It is traditionally named **self**, but this is just a convention.

To construct a new cookie-jar object, we "call" the class name like a function. This allocates the space for a new object and calls the initialization method **__init__** on it.

```
>>> jar = CookieJar('Cookie monster', 7)
>>> dir(jar)
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'add',
 'capacity',
 'eat_cookie',
 'n_cookies',
```

```
 'owner']
>>> jar.n_cookies
0
>>> jar.add(3)
>>> jar.n_cookies
3
```

We can access members from "outside" using **<object>.<member>** syntax. The call **jar.add(3)** is just a shorthand for **CookieJar.add(jar, 3)**.

Let's eat some cookies!

```
>>> jar.eat_cookie()
>>> jar.n_cookies
2
>>> jar.add(10)
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-133-9f0f015dbf6c> in <module>
----> 1 jar.add(10)

<ipython-input-100-e2eeb1c6d7bd> in add(self, n)
      6
      7     def add(self, n):
----> 8         assert self.n_cookies+n <= self.capacity, 'Too many cookies
    \N{drooling face}'
      9         self.n_cookies += n
     10

AssertionError: Too many cookies
>>> jar.eat_cookie()
>>> jar.eat_cookie()
>>> jar.eat_cookie()
>>> jar.eat_cookie()
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-137-0ec055dd03c7> in <module>
----> 1 jar.eat_cookie()

<ipython-input-100-e2eeb1c6d7bd> in eat_cookie(self)
     10
     11     def eat_cookie(self):
---> 12         assert self.n_cookies >= 0, 'NO MORE COOKIES!!!'
     13         self.n_cookies -= 1
     14         print('Om Nom Nom Nom')

AssertionError: NO MORE COOKIES!!!
```

Let's see what happens when we print a cookie jar:

```
>>> print(jar)
<__main__.CookieJar at 0x7f9e13903790>
```

This calls **str(jar)** which by default prints the object's name and address. We can add custom string conversion by defining an **__str__** method:

```
class CookieJar:
    def __init__(self):
        ...
    .
    .
    .
    def __str__(self):
        heading = f"=== {self.owner}'s cookie jar ==="
```

```
        cookies = '[' + '\N{cookie}'*self.n_cookies + '
    '*(self.capacity-self.n_cookies) + ']'
        return '\n'.join([heading, cookies])
```

Now we can get nice prints.

```
>>> print(jar)
>>> jar.add(3)
>>> print(jar)
=== Cookie monster's cookie jar ===
[OOO    ]
```

We can add a **repr** method:

```
    def __repr__(self):
        return f"CookieJar('{self.owner}', {self.capacity},
    {self.n_cookies})"
```

What are the advantages of using a Python class over plain functions as in Section 18.1?

- The methods are implicitly bound to the object – calling **jar.add(n)** implicitly passes the object as the **self** argument to the method.
- We can define special methods such as **__str__** and **__repr__** that other parts of Python can use in a standard way. Another example is **__getstate__** and **__setstate__** which are used by the **pickle** module to save and load our object.
- **Encapsulation** Users of the class don't have to be aware of everything that the class does. For example, if we decide to add a log of cookie eating times this can be "hidden" inside the object without changing the behavior.
- **Reliability** objects often have some invariants that need to be maintained for them to be valid. Classes can help us by guaranteeing that this invariant is maintained by each one of the methods. If the modifications to the objects are only handled by the methods we can reliably guarantee that the object remains valid. In code that passes around dictionaries/tuples/lists, over time they tend to be modified in multiple locations by different developers. This can break the invariants.

"private" members In Python, users can see and modify all the members. The convention is that members whose name starts with an underscore are not to be touched outside of the methods.

18.3 Two-dimensional vectors

Python classes can be used to define how the objects behave when they are acted upon by operators such as **+**, *****, etc. Using methods with special names such as **__add__**. To see an example of this, we will implement a 2D vector class:

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def __repr__(self):
        return f'Vector2D({self.x}, {self.y})'
```

```
    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        return Vector2D(self.x * scalar, self.y * scalar)
```

Using the class:

```
>>> v = Vector2D(1,3)
>>> v
Vector2D(1, 3)
>>> v.norm()
3.1622776601683795
>>> u = Vector2D(10,20)
>>> u+v
Vector2D(11, 23)
>>> v*1000
Vector2D(1000, 3000)
>>> 100*v
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 100*v

TypeError: unsupported operand type(s) for *: 'int' and 'Vector2D'
```

Why did this error occur? Because **__mul__** is called when you multiply a **Vector2D** (on the left) by a scalar (on the right). What we tried to do is multiply a scalar by a Vector2D. When **100*v** is called, Python calls **100.__mul__(v)** which fails because the **int** class does not support **Vector2D**. We cannot change the the definition **int** type, so what can we do? Python's solution is to define a reverse-mul method:

```
class Vector2D:
    .
    .
    .
    def __rmul__(self, scalar):
        return Vector2D(self.x * scalar, self.y * scalar)
```

This resolves the problem:

```
>>> v = Vector2D(1,3)
>>> 100*v
Vector2D(100, 300)
```

An better solution would be to call **Vector2D.__mul__** from inside **Vector2D.__rmul__**:

```
class Vector2D:
    .
    .
    .
    def __rmul__(self, scalar):
        return self*scalar
```

18.3.1 Order relations

Let's try to sort some vectors:

```
>>> u = Vector2D(2,-1)
>>> v = Vector2D(1,3)
```

```
>>> sorted([v,u])
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[42], line 1
----> 1 v < u

TypeError: '<' not supported between instances of 'Vector2D' and 'Vector2D'
```

This fails because we did not define when a **Vector2D** is smaller than another. We can correct this by defining the less-than method:

```
class Vector2D:
    .
    .
    .
    def __lt__(self, other):
        return self.x <= other.x
```

Now we can compare vectors and sort them:

```
>>> u = Vector2D(2,-1)
>>> v = Vector2D(1,3)
>>> u<v
False
>>> sorted([v,u])
[Vector2D(1, 3), Vector2D(2, -1)]
```

Chapter 19. Namespaces

A namespace is a dictionary-like structure that maps names to objects. For example, every module has its own namespace whose keys are the names of all functions, classes and variables defined in that module.

[module namespace]

```
>>> import math
>>> math.__dict__
{'__name__': 'math',
 '__doc__': 'This module provides access to the mathematical
    functions\ndefined by the C standard.',
 .
 .
 .
 'sin': <function math.sin(x, /)>,
 'sinh': <function math.sinh(x, /)>,
 'sqrt': <function math.sqrt(x, /)>,
 'tan': <function math.tan(x, /)>,
 'tanh': <function math.tanh(x, /)>,
 'sumprod': <function math.sumprod(p, q, /)>,
 'trunc': <function math.trunc(x, /)>,
 'prod': <function math.prod(iterable, /, *, start=1)>,
 'perm': <function math.perm(n, k=None, /)>,
 'comb': <function math.comb(n, k, /)>,
 'nextafter': <function math.nextafter(x, y, /, *, steps=None)>,
 'ulp': <function math.ulp(x, /)>,
 '__file__':
    '/opt/anaconda3/lib/python3.12/lib-dynload/math.cpython-312-darwin.so',
 'pi': 3.141592653589793,
 'e': 2.718281828459045,
 'tau': 6.283185307179586,
 'inf': inf,
 'nan': nan}
```

Using **<module>.<name>** is just shorthand for **<module>.__dict__[<name>]**:

```
>>> math.pi
3.141592653589793
>>> math.__dict__['pi']
3.141592653589793
```

Chapter 20. Exceptions

In Python, errors are typically handled using the *exceptions* mechanism. For example, if you access a list at an invalid index, you get an **IndexError**:

```
>>> print(lis[10])
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 print(lis[10])

IndexError: list index out of range
```

Python provides very informative error messages:

- The name of the exception: **IndexError**
- A description: **list index out of range**
- The location where the exception occured: **----> 1 print(lis[10])**

[traceback]

If the exception occured inside a function, Python will print, in addition to the line where the exception was raised, the line where the current function was called, its caller, its caller's caller, etc. This print-out is called a *traceback*. Let's see it in action:

```
def convert_to_int(s):
    return int(s)

def is_string_num_even(s):
    n = convert_to_int(s)
    return (n % 2) == 0
```

The function **is_string_num_even** expects a numeric string:

```
>>> is_string_num_even('100')
True
>>> is_string_num_even('101')
False
```

When we call it with a non-numeric string an exception is raised at the point of the error:

```
>>> is_string_num_even('one hundred')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[17], line 1
----> 1 is_string_num_even('one hundred')

Cell In[15], line 2, in is_string_num_even(s)
      1 def is_string_num_even(s):
----> 2     n = convert_to_int(s)
      3     return (n % 2) == 0
```

```
Cell In[14], line 2, in convert_to_int(s)
      1 def convert_to_int(s):
----> 2     return int(s)

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

We see that the error occured inside **convert_to_int(s)** which was called by the function **is_string_num_even(s)**, called by the command **is_string_num_even('one hundred')** that we typed in the interpreter.

20.1 Catching exceptions

When an exception is raised, Python's default behavior is as follows:

1. First, any resources that were opened within an active **with** block, such as open files, will be closed.
2. Then an error message and traceback will be printed.
3. Finally, the program will be terminated. If you are running code in the Python interpreter, you will find yourself back in the interpreter.

In some cases, we would like to handle the exception and let the program continue. Suppose for example that we wish to process text files. If an error occurs in one of the files, we want to continue processing the other files.

```
>>> def process_text_file(filename):
...     data = open(filename).read()
...     print(f'{filename}: {data[:50]}')
...
>>> for fn in ['doesnotexist.txt', 'somefile.txt']:
...     process_text_file(fn)
...
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-17-0d1fbf5ffa70> in <module>
      1 for fn in ['somefile.txt', 'doesnotexist.txt']:
----> 2     process_text_file(fn)
      3

<ipython-input-16-85b9976b89a0> in process_text_file(filename)
      1 def process_text_file(filename):
----> 2     data = open(filename).read()
      3     print(f'{filename}: {data[:50]}')
      4
      5

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

Let's handle this exception with **try/except**:

```
>>> for fn in ['doesnotexist.txt', 'somefile.txt']:
...     try:
...         process_text_file(fn)
...     except FileNotFoundError:
...         print(f'File "{fn}" not found. Skipping...')
...
File "doesnotexist.txt" not found. Skipping...
somefile.txt: This is the contents of somefile.txt
```

When catching an exception, we can give it a name and get more details about it:

```
>>> for fn in ['somefile.txt', 'doesnotexist.txt']:
...     try:
...         process_text_file(fn)
...     except FileNotFoundError as e:
...         print(f'File "{fn}" not found. Skipping...')
...         print('Error:', e)
...
File "doesnotexist.txt" not found. Skipping...
Error: [Errno 2] No such file or directory: 'doesnotexist.txt'
somefile.txt: This is the contents of somefile.txt
```

20.2 Catching multiple exceptions

```
try:
    num = int(input("Enter a number: "))
    result = 100 / num
except ValueError:
    print("Please enter a valid number")
except ZeroDivisionError:
    print("Cannot divide by zero")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

20.3 Defining your own exceptions

It's easy to define your own exceptions:

```
>>> class NonAlphanumericError(Exception):
...     pass
...
>>> for word in ['one', 'two', 'encyclopedia', 'uygr436#']:
...     if not word.isalnum():
...         raise NonAlphanumericError(word)
...     print(word)
...
---------------------------------------------------------------------------
NonAlphanumericError                        Traceback (most recent call
    last)
<ipython-input-25-0ed8005e5a39> in <module>
      1 for word in ['one', 'two', 'encyclopedia', 'uygr436#']:
      2     if not word.isalnum():
----> 3         raise NonAlphanumericError(word)
      4     print(word)
      5

NonAlphanumericError: uygr436#
```

20.4 pdb.pm

When an exception is raised but not caught by the code, it is caught by the Python interpreter and a nice traceback is called.

```
>>> for fn in ['somefile.txt', 'doesnotexist.txt']:
...     process_text_file(fn)
...
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-42-35acd58d4984> in <module>
      1 for fn in ['somefile.txt', 'doesnotexist.txt']:
----> 2     process_text_file(fn)
      3

<ipython-input-16-85b9976b89a0> in process_text_file(filename)
      1 def process_text_file(filename):
----> 2     data = open(filename).read()
      3     print(f'{filename}: {data[:50]}')
      4
      5

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

We can use the built-in Python debugger **pdb** to jump to where the error occured.

```
>>> import pdb
>>> pdb.pm()
> <ipython-input-16-85b9976b89a0>(2)process_text_file()
-> data = open(filename).read()
(Pdb) l
  1     def process_text_file(filename):
  2  ->     data = open(filename).read()
  3         print(f'{filename}: {data[:50]}')
  4
  5
[EOF]
(Pdb) ?

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb) ? p
p expression
        Print the value of the expression.
(Pdb) p filename
'blah.txt'
(Pdb) up
> <ipython-input-42-35acd58d4984>(2)<module>()
-> process_text_file(fn)
(Pdb) l
  1     for fn in ['somefile.txt', 'doesnotexist.txt']:
  2  ->     process_text_file(fn)
  3
[EOF]
(Pdb) q
```

Part III

Python for Data Science

Chapter 21. Bits and bytes

21.1 Bitwise operations

In Python, numbers can be writte in base 2 as follows:

```
>>> 0b101
5
```

We can apply bitwise operations **&,|,^** to any **int**:

```
>>> 27&12
8
```

[bin] Why did we get this result? Let's see these numbers in binary:

```
>>> bin(27)
'0b11011'
>>> bin(12)
'0b1100'
>>> bin(8)
'0b1000'
```

Since these binary numbers don't have the same number of digits, they are not properly aligned. We can fix this with f-strings:

```
\begin{codebox}
>>> for num in [27, 12, 27&12]:
...     print(f'{num:05b}')
...
11011
01100
01000
```

You can use the **0b** (binary) prefix to write integers in base 2:

```
>>> a = 0b1011
11
>>> type(a)
int
>>> b = 13
>>> a^b
6
>>> bin(a^b)
'0b110'

>>> print(f'Decimal: {a}^{b} == {a^b}    Binary: {a:04b}^{b:04b}=={a^b:04b}')
Decimal: 13^11 == 6    Binary: 1101^1011==0110
```

```
>>> a = 0b1101
>>> a
13
>>> b = 0b1011
>>> b
```

```
11
>>> c = a^b
>>> print(f'{a}^{b} == {c} {a:04b}^{b:04b}=={c:04b}')
>>> print(f'{a}^{b} == {c}    {a:04b}^{b:04b}=={c:04b}')
    a = 0b1101  # Binary for 13
b = 0b1011  # Binary for 11
print(a, b)  # Prints: 13 11
```

You can use f-strings to print numbers in binary:

```
print(f"{a} in binary is {a:04b}")  # Pads to 4 bits: 1101
```

21.2 Calculating parity

The parity of a number is the number of 1s in its binary representation. Let's write some code that calculates it:

```
def parity(n):
    p = 0
    while n:
        p ^= 1
        n &= n-1
    return p
```

If we know that our number has a limited number of bits, we can use a bit mask to calculate the parity:

```
>>> x = 0b110101
>>> (x&0b01010101) ^ (x&0b10101010)
```

TODO: Continue

21.3 Bit sequences as subsets

A sequence of n bits can be used to represent a subset of a set of n elements. Each bit corresponds to an element in the set: if the bit is 1, the element is included in the subset; if the bit is 0, it is excluded. For example, we can represent the subsets of $S = \{a, b, c, d\}$ using a 4-bit binary number, where each bit position corresponds to an element in S.

Binary	Subset	Decimal
0000	$\emptyset$	0
0001	$\{d\}$	1
0010	$\{c\}$	2
0011	$\{c, d\}$	3
0101	$\{b, d\}$	5
1111	$\{a, b, c, d\}$	15

21.4 Bitwise operations and their set equivalents

Python provides bitwise operators that align naturally with set operations:

Bitwise Operation	Python Syntax	Set Operation
Complement	`~S`	S^c
And	`S&T`	$S \cap T$
Or	`S|T`	$S \cup T$
Xor	`S^T`	$S \Delta T$

21.5 Example: iterating over subsets

Example. Given an input sequence of numbers, find a subsequence who's sum equalt a certain value.

```
def exact_subseq_sum(numbers, target):
    for bits in range(1 << len(numbers)):
        indices = [i for i in range(len(numbers)) if bits&(1<<i)]
            if sum([numbers[i] for i in indices]) == target:
            return indices
```

Faster algorithm using Grey code:

```
def exact_subseq_sum_grey(numbers, target):
log2dict = {2**i: i for i in range(len(numbers))}
prev_bits = 0
cur_sum = 0
for i in range(1, 1<<len(numbers)):
    bits = i^(i>>1)
    #print(f'Current bits: {bits:08b}')
    bits_diff = bits^prev_bits
    #print(f'bits difference: {bits_diff:08b}')
    diff_index = log2dict[bits_diff]
    #print(f'Index of difference: {diff_index}')
    if bits_diff&bits:
        cur_sum += numbers[diff_index]
    else:
        cur_sum -= numbers[diff_index]
    if cur_sum == target:
        return [i for i in range(len(numbers)) if bits&(1<<i)]
    #print(f'Current sum: {cur_sum}')
    #print()
    prev_bits = bits
```

Chapter 22. Binary I/O

22.1 bytes

[bytes]

Python defines a **bytes** object that is similar to a string, but stores a sequence of bytes. We can initialize it from a sequence of numbers (e.g. a list or tuple).

```
>>> bytes([6,5,4])
b'\x06\x05\x04'
>>> bytes(range(5))
b'\x00\x01\x02\x03\x04'
>>> bytes([1000])
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In, line 1
----> 1 bytes([1000])
ValueError: bytes must be in range(0, 256)
```

bytes objects are displayed as a string prefixed by the letter **b**. Each byte is written as **\x** followed by a two-digit hexadecimal representation. However, if the byte corresponds to a printable ASCII character, it is displayed as the ASCII character.

```
>>> bytes([72, 101, 108, 108, 111])
b'Hello'
```

This can get confusing if we mix printable and non-printable bytes:

```
>>> bytes(range(256))
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13
\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./0123456789:
;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~\x7f'
```

They can be initialized from **b**-strings, like this:

```
>>> x = b'Hi Ther\x65\x21'
>>> print(x)
b'Hi There!'
```

They can be indexed, sliced and concatenated:

```
>>> x[2]
32
>>> x[1:5]
b'i Th'
>>> x[0::2] + x[1::2]
b'H hr!iTee'
```

Note the quirk that indexing a **bytes** object returns an integer in the range 0–255, but taking a slice returns a **bytes**object. Unlike strings, the values used to initialize it must be numbers in the range $0, \ldots, 255$ or you will get an error:

```
>>> bytes((100,1000))
---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call last)
----> 1 bytes((100,1000))

ValueError: bytes must be in range(0, 256)
```

22.2 Reading binary files

[open] Let's read a PNG image file. First we need to open the file:

```
>>> f = open('cute.bmp','rb') # rb = Read Binary file
```

The **open** command returns a *file handle*, that is an object used to manipulate an open [read] file. Let's read some bytes:

```
>>> f = open('cute.png', 'rb')
>>> f.read(10)
b'BM6\x10\x0e\x00\x00\x00\x00\x00'
>>> f.read(10)
b'6\x00\x00\x00(\x00\x00\x00\xe0\x01'
```

The command **f.read(n)** tells the file object to read **n** bytes from the file and return them as a **bytes** object. The current position in the file is automatically moved to the end of what was read. We can see that this BMP file starts with 'BM'. This is part of the BMP file header:

Offset	Size	Type	Field
0	2	-	Signature ('BM')
2	4	uint32	File size
6	2	-	Reserved1
8	2	-	Reserved2
10	4	uint32	Pixel data offset

We see that the first field is just 'BM'. The second field is the file size as a uint32. Let's [] read it, but first we need to move the reading position to 2 using **f.seek(2)**: seek]

```
>>> f.seek(2)
2
>>> filesize_field = f.read(4)
>>> filesize_field
b'6\x10\x0e\x00'
```

The numbers in a BMP file are stored in little-endian order, so we can parse it as follows:

```
>>> filesize_field[0] | (filesize_field[1]<<8) + (filesize_field[2]<<16) +
    (filesize_field[3]<<24)
921654
```

This is indeed the file size! We can read the rest of this file using **f.read()**:

```
>>> data = f.read()
>>> len(data)
921648
```

We will get back to this example later.

22.2.1 Writing binary files

To open a file for writing, use the same **open** command, but with a parameter **'wb'** (short for write-binary):

```
>>> f = open('hi', 'wb')
```

[write] If the file **hi** already exists it will be overwritten. Let's try to write a message:

```
>>> f.write('Hey...')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
----> 1 f.write('Hey...')

TypeError: a bytes-like object is required, not 'str'
```

Why doesn't this work? Because we are trying to write a string object to a binary file but Python expects a sequence of bytes.

```
>>> f.write(b'Hey...')
6
```

If you now try to read the file it will appear empty:

```
>>> open('hi','rb').read()
b''
```

Why? For efficiency reasons, calls to **.write()** fill a buffer in memory and occassionally write it out to disk. There are 3 ways to make sure the data is actually written to disk:

[flush]
- (less common) Call **.flush()** to write all the data that's currently buffered.

[close]
- (more common) Call **.close()** to flush the data and close the file.

[with]
- Use a **with** block to open the file. This will automatically close the file when the block is exited or when an exception is raised.

The recommended method is to use a **with** block when writing:

```
>>> with open('hi', 'wb') as f:
...     f.write(b'Hey...')
...
>>> open('hi','rb').read()
b'Hey...'
```

Let's close the file and read back what we wrote:

```
>>> f.close()
>>> open('hi','rb').read()
b'Hey...'
```

22.3 Example: appending a secret message

You can stick extra contents at the end of a BMP file (and most files) with no issues whatsoever.

```
>>> def add_secret_message(infile, outfile, secret_bytes):
...     indata = open(infile, 'rb').read()
...     with open(outfile, 'wb') as f:
...         f.write(indata)
...         f.write(secret_bytes)
...
>>> add_secret_message('cute.bmp', 'cute-secret.bmp', b'[[[A VERY SECRET
    SECRET]]]')
```

Another way is to append to an existing file. Let's first make a copy, in case we make any mistakes:

```
>>> import shutil
>>> shutil.copy('cute.bmp', 'cute2.bmp')
'cute2.bmp'
```

Now we can append the message.

```
>>> def append_secret_message(filename, secret_bytes):
...     with open(filename, 'ab') as f:
...         f.write(secret_bytes)
...
>>> append_secret_message('cute2.bmp', b'[[[SECRET SECRET]]]')
>>> open('cute2.bmp','rb').read()[-100:]
b'^ekW_eXafYafYaf^fj]di[bgZ`eY_dZ^cX]bX]b[`e]bg]`e\\_c^ae\\_aX[_X[]X[_X[]Y[]Y[]WY[UXY[[[SECRET
    SECRET]]]'
```

22.4 Example: parsing a BMP file

We've already seen the first bitmap header. It is followed by another header, called DIB. Here is its structure (with some irrelevant final fields omitted):

Offset	Size	Type	Field
14	4	uint32	DIB Header size (=40)
18	4	int32	Width
22	4	int32	Height
26	2	uint16	#Planes (must be 1)
28	2	uint16	Bits per pixel (24 for RGB image)
30	4	uint32	Compression method (0 for uncompressed)

Let's write a parser for BMP files!

```
    import struct
    import numpy as np

    def read_bmp_header(f):
        """Reads BMP and DIB headers, returns metadata and pixel data
    offset."""
        file_header = f.read(14)
        signature, file_size, reserved1, reserved2, pixel_offset =
    struct.unpack('<2sIHHI', file_header)

        if signature != b'BM':
            raise ValueError("Not a valid BMP file")

        dib_header = f.read(40)
        (header_size, width, height, planes, bpp, compression, image_size,
         x_ppm, y_ppm, colors_used, important_colors) =
    struct.unpack('<IIIHHIIIIII', dib_header)

        if bpp != 24 or compression != 0:
            raise ValueError("Only uncompressed 24-bit BMP files are
    supported")
        return (width, height, pixel_offset)
```

```
    def read_bgr_row(row_data, width):
        """Efficiently extracts R, G, B lists from a BGR row using slices
    only."""
        pixels = row_data[:width * 3]  # Strip padding
        R = list(pixels[2::3])
        G = list(pixels[1::3])
        B = list(pixels[0::3])
        return R, G, B

    def read_bmp_24bit_channels(filepath):
        with open(filepath, 'rb') as f:
            (width, height, pixel_offset) = read_bmp_header(f)

            f.seek(pixel_offset)
            row_padded = (width * 3 + 3) & ~3

            R = np.zeros((height, width), dtype=np.uint8)
            G = np.zeros((height, width), dtype=np.uint8)
            B = np.zeros((height, width), dtype=np.uint8)

            for y in range(height):
                row_data = f.read(row_padded)
                r_row, g_row, b_row = read_bgr_row(row_data, width)
                row_idx = height - 1 - y  # bottom-up storage
                R[row_idx, :] = r_row
                G[row_idx, :] = g_row
                B[row_idx, :] = b_row

            return R, G, B

    def show_img(R, G, B):
        import matplotlib.pyplot as plt
        rgb_image = np.stack((R, G, B), axis=2)
        plt.imshow(rgb_image)
        plt.show()
```

Chapter 23. Field packing

Recall that before we converted a 4-byte sequence to an unsigned integer as follows:

```
>>> heightbin = bytes([0,0,5,0])
>>> 256**3 * heightbin[0] + 256**2 * heightbin[1] + 256**1 * heightbin[2] +
    256**0 * heightbin[0]
1280
```

[int.from_bytes] Python has a built-in function called **int.from_bytes** that does the same thing:

```
>>> int.from_bytes(heightbin, byteorder='big')
1280
```

23.1 struct

[struct] Another way to decode integers is to use the **struct** module:

```
>>> struct.unpack('>L', heightbin)
(1280,)
```

The first parameter to **unpack** is the format parameter. **'>L'** means big-endian uint32 (4-byte unsigned integer). The **struct** module can be used to decode and also encode multiple fields. Here's an example that decodes a byte string as a sequence of two uint8 (unsigned byte), one uint16 (an unsigned 2-byte integer) and one uint32 (an unsigned 4-byte integer), all in big endian order.

```
>>> b = bytes([10,20,30,40,50,60,70,80])
>>> b
b'\n\x14\x1e(2<FP'
>>> struct.unpack('<BBHL', b)
(10, 20, 10270, 1346780210)
```

Let's encode the numbers we decoded in big-endian format:

```
>>> struct.pack('>BBHL', 10, 20, 10270, 1346780210)
b'\n\x14(\x1ePF<2'
```

The struct module is very helpful to parse binary files if you know their format. You can learn more about the **struct** module here: https://docs.python.org/3.11/library/struct.html

Chapter 24. Unicode

24.1 Unicode, strings and bytes

The Unicode standard assigns a number to each character called a *code point*. To get
[ord] the code point of a character, use the **ord** function:

```
>>> ord('a')
97
>>> ord('👻')
128123
```

[chr] The reverse function is **chr**:

```
>>> chr(128123)
'👻'
```

See https://home.unicode.org to learn more about unicode and discover exciting facts such as the most frequently-used emoji.

24.2 Converting between strings and bytes

In Python 3, strings are sequences of unicode characters. However, the computer's memory and disk can only store sequences of bytes. To convert between a string and its byte representation, we need to use the **.encode** function that converts a string to a
[UTF-8] **bytes** object:
[UTF-32]

```
>>> 'Hi! 💩'.encode('utf-8')
b'Hi! \xf0\x9f\x92\xa9'
>>> 'Hi! 💩'.encode() # UTF-8 is the default
b'Hi! \xf0\x9f\x92\xa9'
>>> 'Hi! 💩'.encode('utf-32')
b'\xff\xfe\x00\x00H\x00\x00\x00i\x00\x00\x00!\x00\x00\x00
    \x00\x00\x00\xa9\xf4\x01\x00'
>>> b'\xff\xfe\x00\x00H\x00\x00\x00i\x00\x00\x00!\x00\x00\x00
    \x00\x00\x00\xa9\xf4\x01\x00'.decode('utf-32')
'Hi! 💩'
```

UTF-8 and UTF-32 are two examples of unicode encoding formats. UTF-32 always uses 32 bits (=4 bytes) to store each unicode character whereas UTF-8 uses a variable-length encoding that takes up less space. Python's default is to use UTF-8 everywhere. Use the **bytes.decode** function to convert a **bytes** object back to a string:

```
>>> b'\xff\xfe\x00\x00H\x00\x00\x00i\x00\x00\x00!\x00\x00\x00
    \x00\x00\x00\xa9\xf4\x01\x00'.decode('utf-32')
'Hi! 💩'
```

Chapter 25. Text file encodings

Python has full unicode support. Let's see if we can save a "merperson" (a non-binary mermaid) emoji to a file:

```
>>> with open('tmp.txt', 'w') as f:
...     f.write('\N{merperson}')
...
>>> print(open('tmp.txt').read())
```

[encoding]

If you look at the file that was created you will see that it is 4 bytes long. But why? How is the unicode symbol "merperson" encoded into bytes? This is the job of a *unicode encoding*. Check the default encoding in your system like this:

```
>>> import sys
>>> sys.getfilesystemencoding()
'utf-8'
```

UTF-8 is usually the best choice. To enforce it we set the **encoding** argument **open**:

```
>>> with open('tmp.txt', 'w', encoding='utf-8') as f:
...     f.write('\N{merperson}')
...
>>> print(open('tmp.txt').read())
>>> print(open('tmp.txt', encoding='utf-8').read())
```

Chapter 26. NumPy

NumPy is a library for working with homogeneous arrays and matrices – that is, arrays where all elements have the same type. Compared to Python lists, NumPy arrays have a smaller memory footprint and can be used to write much faster code.

26.1 Importing NumPy

If you installed the Anaconda Python package, NumPy is already included. We'll start by importing the **numpy** library:

```
>>> import numpy
>>> numpy.__version__
'1.21.5'
```

If you try to run **dir(numpy)** you'll learn that this module has hundreds of functions. Many of them are quite useful! To save ourselves the trouble of typing **numpy** every time, it is customary to use the shorthand **np**.

[import as]

```
>>> import numpy as np
```

This is what we'll do from now on.

26.2 numpy.array

The **array** class is the main workhorse of the NumPy module. It can represent vectors, matrices and n-dimensional arrays of various data types. We'll start by initializing a one-dimensional integer array from a list.

[array]

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

Numpy arrays are *typed*. Since our initializer list **[1,2,3]** contained only integers, NumPy deduced that we want to construct an array of type **int**. Trying to store an incompatible type gives an error:

```
>>> a[1] = 'hi'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-87732715c108> in <module>
----> 1 a[1] = 'hi'

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

What if we try to store a **float** value?

```
>>> a[1] = 1.8
>>> a
array([1, 1, 3])
```

Python casts the value to **int**, as if we did **a[1] = int(1.8)**. We can check the data type of an array using **.dtype**

[dtype]

```
>>> a = np.array([1,2,3])
>>> a.dtype
dtype('int64')
```

If the initializing list contains floating-point numbers, numpy will use a float dtype,

```
>>> b = np.array([1, 2, 3.1])
>>> b.dtype
dtype('float64')
```

We can also specify the dtype explicitly during construction:

```
>>> np.array([1,2,3], dtype=np.int64)
array([1, 2, 3])
>>> np.array([1,2,3], dtype=np.float64)
array([1., 2., 3.])
```

The **dtype** can be used to specify lower-precision floating-point numbers: **float16, float32**. Lower-range integers: **int8, int16, int32** and unsigned integers are also supported: **uint8, uint16, uint32, uint64**.

[astype] You can convert an array to one with a different dtype using **.astype(dtype)**

```
>>> arr = np.array([1, 2])
>>> arr.dtype
dtype('int64')
>>> arr.astype(np.float64)
array([1., 2.])
```

This creates a copy and does not modify the original array.

Remark 5. Note on dtype

By default, NumPy's constructors return an array of dtype **numpy.float_** (with a trailing underscore). On most systems this is the *double-precision floating-point* type **numpy.float64** that is compatible with Python's **float** and C's **double** type. Your CPU has components dedicated to fast floating-point arithmetic called FPU (Floating-Point Unit). They can work in either single precision using 32-bit floats or double-precision 64-bit floats. Double-precision floats are recommended for general use. Single-precision floats can be specified by setting **dtype** to **numpy.float32**. They are used in specialized settings such as storing huge data sets or for slightly faster computations in cases where the precision is not critical.

26.3 One-dimensional array creation

[np.zeros] To create an array of zeros, use **np.zeros**

```
>>> np.zeros(10)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> np.zeros(10, dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[no.ones] Guess how to create an array of ones...

```
>>> np.ones(7)
array([1., 1., 1., 1., 1., 1., 1.])
```

[np.arange] The **np.arange** function creates an integer array from start/stop/step arguments, similar to the built-in **range** command.

```
>>> np.arange(0,50,5)
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])
```

[np.linspace] To create an evenly-spaced floating-point sequence, use **np.arange**:

```
>>> np.linspace(0,10,4)
array([ 0.        ,  3.33333333,  6.66666667, 10.        ])
```

26.4 Array operations

Vector-scalar operations work element-wise. Let's see some examples:

```
>>> X = np.linspace(0,10,4)
>>> X
array([ 0.        ,  3.33333333,  6.66666667, 10.        ])
>>> X*2
array([ 0.        ,  6.66666667, 13.33333333, 20.        ])
>>> X + 7
array([ 7.        , 10.33333333, 13.66666667, 17.        ])
>>> X + np.arange(4)
array([ 0.        ,  4.33333333,  8.66666667, 13.        ])
```

Even trigonometric functions are supported:

```
>>> X = np.linspace(0,2*np.pi, 100)
>>> Y = np.sin(X)
```

The NumPy module has many other functions that work on arrays, including: **sum**, **mean**, **min**/**max**, **argmin**/**argmax**, **var**, **std**, etc. To learn more, check out the official NumPy docs at https://numpy.org/

26.5 Speed comparison

Working with numpy arrays in a vectorized style is much faster than plain Python computations.

```
>>> X = list(range(1000))
>>> %timeit -n1000 Y=[x**2 for x in X]
In [167]: %timeit -n1000 Y = [x**2 for x in X]
25.4μs ± 4.12μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

vs.

```
>>> X = np.arange(1000)
>>> %timeit -n1000 Y=X**2
1.05 μs ± 99.3 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Almost 25 times faster!

26.6 Creating 2D arrays

We can initialize a 2-dimensional array from a list of lists

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

Each of the sublists is a row of the new 2-dimensional array. It is our responsibility to make sure all the rows are the same length. The array **a** has a **shape** 2×3. This can be accessed using the **.shape** member

[shape]

```
>>> a.shape
(2, 3)
```

Array access is done using **[row,col]** subscripting:

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

The functions **zeros**, **ones** can take a **shape** argument

```
>>> np.zeros((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.ones((3,2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

The **array.reshape** method can be used to change the shape of an existing array. This can be used for initialization:

```
>>> np.arange(12).reshape((3,4))
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

26.7 Array slicing

Arrays can be sliced using **[start:stop:step]** notation, just like lists. However, unlike lists, slices are references, or *views* into an existing array!

```
>>> A = np.arange(10)
>>> B = A[::2]
>>> A
array([0, 1, 0, 3, 0, 5, 0, 7, 0, 9])
>>> B
array([0, 2, 4, 6, 8])
>>> B[:] = 0
>>> B
array([0, 0, 0, 0, 0])
>>> A
array([0, 1, 0, 3, 0, 5, 0, 7, 0, 9])
```

To prevent this behavior you can use **.copy()**

```
>>>  A = np.arange(10)
>>> B = A[::2].copy()
>>> B[:] = 0
>>> A
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> B
array([0, 0, 0, 0, 0])
```

Multidimensional arrays can be sliced too:

```
>>> C = np.arange(50).reshape((5,10))
>>> C
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
>>> C[1:-1, 2:4]         # Drop the top and bottom rows and take columns 2,3
array([[12, 13],
       [22, 23],
       [32, 33]])
>>> C[1:,:]              # Drop the first row
array([[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
>>> C[1:-1, 1:-1] += 100 # Add 100 to all cells on the inside of the matrix
>>> C
array([[  0,   1,   2,   3,   4,   5,   6,   7,   8,   9],
       [ 10, 111, 112, 113, 114, 115, 116, 117, 118,  19],
       [ 20, 121, 122, 123, 124, 125, 126, 127, 128,  29],
       [ 30, 131, 132, 133, 134, 135, 136, 137, 138,  39],
       [ 40,  41,  42,  43,  44,  45,  46,  47,  48,  49]])
```

We can extract a single row or column by slicing:

```
>>> C[1,:]
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>> C[:,3]
array([ 3, 13, 23, 33, 43])
```

Note that the returned array is one-dimensional in both cases. An equivalent method to access a row is by a single subscript

```
>>> C[1]
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

This mimics the list-of-list structure. If you want to treat a column as a 2-dimensional array of width 1 you can use reshape:

```
>>> C[:,3].reshape((5,1))
array([[ 3],
       [13],
       [23],
       [33],
       [43]])
```

26.8 Concatenating arrays

[np.hstack] To concatenate arrays, we can use the **hstack** (horizontal stack) function:

```
>>> A
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.hstack([A,A])
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[np.vstack] Let's try stacking vertically:

```
>>> np.vstack([A,C])
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

This works even though **A** is one-dimensional and **C** is two-dimensional.

26.9 asarray

It is common to write a function that takes a NumPy array but can also accept lists or tuples. We could initialize a new **numpy.array** from the input, but that would be wasteful if the input is already an array. This is what the **numpy.asarray** function is for. It constructs a new **numpy.array** but only if the input object is not already an [np.asarray] array. It also receives an optional dtype argument.

```
>>> np.asarray([1,2,3])
array([1, 2, 3])
>>> np.asarray([1,2,3], dtype=np.float64)
array([1., 2., 3.])
```

If the input is already an array, it simply returns a reference to the input.

```
>>> arr = np.arange(5)
>>> arr2 = np.asarray(arr)
>>> arr is arr2
True
```

26.10 Beware of integer overflow

It is important to be aware of the valid range of the different integer types. Unlike Python integers, in NumPy integers can *overflow*. They behave as if the computation are performed modulo 2^b where b is the number of bits. For example, since $2^8 = 256$ the range of a **uint8** is $0, \ldots, 255$.

```
>>> arr = np.arange(256, dtype=np.uint8)
>>> print(arr)
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
```

```
 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251
 252 253 254 255]
>>> print(arr+8)
[  8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25
  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43
  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61
  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97
  98  99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133
 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151
 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169
 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187
 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205
 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241
 242 243 244 245 246 247 248 249 250 251 252 253 254 255   0   1   2   3
   4   5   6   7]
```

The signed integers can also overflow, causing a positive result to turn negative.

```
>>> signedarr = np.arange(128, dtype=np.int8)
>>> print(signedarr)
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127]
>>> print(signedarr+8)
[   8    9   10   11   12   13   14   15   16   17   18   19   20   21
   22   23   24   25   26   27   28   29   30   31   32   33   34   35
   36   37   38   39   40   41   42   43   44   45   46   47   48   49
   50   51   52   53   54   55   56   57   58   59   60   61   62   63
   64   65   66   67   68   69   70   71   72   73   74   75   76   77
   78   79   80   81   82   83   84   85   86   87   88   89   90   91
   92   93   94   95   96   97   98   99  100  101  102  103  104  105
  106  107  108  109  110  111  112  113  114  115  116  117  118  119
  120  121  122  123  124  125  126  127 -128 -127 -126 -125 -124 -123
 -122 -121]
```

To prevent overflow you can either:

- Use an integer type that's bigger than any number you plan to encounter during computations.
- Use floating-point numbers, which have a much larger range of values. The drawback of this is that floating-point numbers typically lose a small amount of accuracy in every arithmetic operation.

26.11 Array internals

After construction, the items in a NumPy array are just a chunck of bytes in memory. We can access it using the **`.data`** member

```
>>> X = np.arange(1,7)
>>> X.data
<memory at 0x7f8c880eba00>
```

```
>>> type(X.data)
memoryview
```

This object represents a buffer in memory. Let's see what methods it exposes:

```
>>> dir(X.data)
['__class__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__len__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'c_contiguous', 'cast', 'contiguous', 'f_contiguous', 'format',
 'hex', 'itemsize', 'nbytes', 'ndim', 'obj', 'readonly', 'release',
 'shape', 'strides', 'suboffsets', 'tobytes', 'tolist', 'toreadonly']
```

We can see the contents of the buffer using **`.tobytes()`**

```
>>> X.data.tobytes()
b'\x01\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x03\x00
\x00\x00\x00\x00\x00\x00 \x04\x00\x00\x00\x00\x00\x00\x00\x05\x00\x00\x00
\x00\x00\x00\x00\x00\x06\x00\x00\x00\x00\x00\x00\x00'
```

The first 8 bytes are **`'\x01\x00\x00\x00\x00\x00\x00\x00'`**. This is just the memory representation of the number **`1`** as an **`int64`** (8-byte integer). The first byte encodes the lowest 8 bits of the number, the second byte encodes bits $8, ..., 15$, ..., etc. Finally, the 8^{th} byte encodes bits $56, ..., 63$. Ordering the bytes in this way is called *little-endian* and this is how most systems work.

If **`X`** is a contiguous one-dimensional array of **`int64`** then when you access **`X[k]`**, NumPy will load the 8 bytes in the data buffer that are at locations $8k, 8k+1, \ldots, 8k+7$. However, if you take **`X2 = X[::2]`**, you get a numpy array that is a view into **`X`**. Accessing $X2[k]$ loads the 8 bytes starting at $16k$. How does NumPy handle all of these cases? Using **`strides`**.

```
>>> arr = np.arange(1,7).reshape((2,3))
>>> arr
array([[1, 2, 3],
       [4, 5, 6]])
>>> arr.strides
(24, 8)
```

The **`strides`** tells NumPy how to interpret indexing operations such as . In this example, **`arr[i,j]`** reads from the buffer at position **`strides[0]*i + strides[1]*j`**. This is because by default, NumPy arrays are stored contiguously row-by-row. Each element has a size of 8 and each row has 3 elements.

Let's see what happens when we reshape the array,

```
>>> arreshaped = arr.reshape((3,2))
>>> arreshaped
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> arreshaped.strides
(16, 8)
```

Reshaping does NOT change the underlying data buffer, it merely changes the strides. Similarly, transposing a matrix merely flips the row and column strides:

```
>>> arrtransposed = arr.T
>>> arrtransposed
array([[1, 4],
```

```
       [2, 5],
       [3, 6]])
>>> arrtransposed.strides
(8, 24)
```

26.12 Row-wise and column-wise aggregation

```
>>> arr
array([[1, 2, 3],
       [4, 5, 6]])
>>> arr.sum()
21
>>> arr.sum(axis=0)
array([5, 7, 9])
>>> arr.sum(axis=1)
array([ 6, 15])
```

Other aggregators that accept an axis parameter: max, min, prod, mean, std, var, all, any, median, argmin, argmax.

26.13 Broadcasting

```
>>> X = np.ones((3,3))
>>> X
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
>>> row = np.arange(3)
>>> row
array([0, 1, 2])
>>> col = row.reshape((3,1))
>>> col
array([[0],
       [1],
       [2]])
>>> row[:,np.newaxis]
array([[0],
       [1],
       [2]])
>>> X + row
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
>>> X + col
array([[1., 1., 1.],
       [2., 2., 2.],
       [3., 3., 3.]])
>>> row+col
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
>>> row*col
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

Example. Centering a data matrix.

```
>>> X = np.arange(50).reshape((5,10))
```

```
>>> X
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
>>> means = X.mean(axis=0)
>>> means
array([20., 21., 22., 23., 24., 25., 26., 27., 28., 29.])
>>> X - means
array([[-20., -20., -20., -20., -20., -20., -20., -20., -20., -20.],
       [-10., -10., -10., -10., -10., -10., -10., -10., -10., -10.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [ 10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.],
       [ 20.,  20.,  20.,  20.,  20.,  20.,  20.,  20.,  20.,  20.]])
```

26.14 Array masking

[mask] We can use comparison operators on an array. The result is a numpy array of the same size of dtype bool.

```
>>> X = np.array([4,5,3,8,1,0,10,-5,3,5,-9,8])
>>> mask = X>0
>>> mask
array([ True,  True,  True,  True,  True, False,  True, False,  True,
        True, False,  True])
```

A boolean array like this is often called a *mask* since it can be used to mask values that don't match a condition.

Masks can be used to index arrays, like this:

```
>>> X[X<0]
array([-5, -9])
```

Example. Fetching all the values that are divisible by 3:

```
>>> X[(X % 3)==0]
array([ 3,  0,  3, -9])
```

Example. Counting the number of positive values:

```
>>> np.sum(X > 0)
9
```

Why does this work? When boolean values are summed they are automatically converted to integers **`False`**→ 0, **`True`**→ 1. So the call **`np.sum(X > 0)`** counts the number of **`True`** values in the mask **`X > 0`**.

Example. Summing the negative values of an array:

```
>>> X<0
array([False, False, False, False, False, False, False,  True, False,
       False,  True, False])
>>> X[X<0]
array([-5, -9])
>>> np.sum(X[X<0])
np.int64(-14)
```

[np.where] *Example.* You can use the **`np.where`** function to fetch the *indices* of the positive values.

```
>>> np.where(X>0)
(array([ 0,  1,  2,  3,  4,  6,  8,  9, 11]),)
```

26.14.1 Masking with 2D arrays

2D arrays are also supported:

```
>>> X = np.array([4,5,3,8,1,0,10,-5,3,5,-9,8]).reshape((4,3))
>>> X
array([[ 4,  5,  3],
       [ 8,  1,  0],
       [10, -5,  3],
       [ 5, -9,  8]])
>>> X > 5
array([[False, False, False],
       [ True, False, False],
       [ True, False, False],
       [False, False,  True]])
```

Example. How many positive values are in each column?

```
>>> (X > 0).sum(axis=0)
array([4, 2, 3])
```

We can index the array using a mask:

```
>>> X[mask]
array([ 4,  5,  3,  8,  1, 10,  3,  5,  8])
```

The result is a one-dimensional copy of the elements for which the condition is true. Complex conditions can be expressed by boolean operators `&`, `|`, `~`

```
>>> (X > 5) & (X < 10)
array([[False, False, False],
       [ True, False, False],
       [False, False, False],
       [False, False,  True]])
>>> np.sum((X > 5) & (X < 10))
2
```

Example. What is the average of the positive values?

```
>>> np.mean(X[X > 0])
5.222222222222222
```

26.15 Fancy indexing

You can index an array by an array of indices:

```
>>> X = np.array([4,5,3,8,1,0,10,-5,3,5,-9,8])
>>> X[[1,0,1,3]]
array([5, 4, 5, 8])
```

This fetches the values of the indices `[1,0,1,3]` and returns the values inside an array. Unlike indexing by a mask, here the elements can be extracted in any order and single element can be extracted multiple times. You can use fancy indexing to modify an array:

```
>>> X[[1,0,1,3]] = [10,20,30,40]
>>> X
array([20, 30,  3, 40,  1,  0, 10, -5,  3,  5, -9,  8])
>>> X[[1,0,1,3]] = 99
>>> X
array([99, 99,  3, 99,  1,  0, 10, -5,  3,  5, -9,  8])
```

26.15.1 numpy.where

One can convert a boolean mask to an index array using the **np.where** function:

```
>>> np.where([False, True, False, False, True])
(array([1, 4]),)
```

Note that this returns a tuple of arrays. *Example.* Extracting even numbers that are at odd indices:

```
>>> np.where((X%2)==0))
(array([ 0,  3,  5,  6, 11]),)
>>> indices = np.where((X%2)==0)[0]
>>> indices[(indices%2) == 1]
array([ 3,  5, 11])
```

26.16 Saving and loading arrays

[np.save] NumPy provides a simple way to save arrays to disk using **save** and **load**:

```
>>> arr = np.array([[1,2,3], [4,5,6]])
>>> np.save('myarray.npy', arr)
>>> loaded_arr = np.load('myarray.npy')
>>> loaded_arr
array([[1, 2, 3],
       [4, 5, 6]])
```

26.17 Basic linear algebra

NumPy provides comprehensive support for linear algebra operations. Let's start with [matrix multiply] matrix multiplication using the `@` operator (introduced in Python 3.5):

```
>>> A = np.array([[1,2], [3,4]])
>>> B = np.array([[5,6], [7,8]])
>>> A @ B  # Matrix multiplication
array([[19, 22],
       [43, 50]])
>>> A * B  # Element-wise multiplication
array([[ 5, 12],
       [21, 32]])
```

The **numpy.linalg** module provides more advanced linear algebra operations. [np.linalg] Let's see some common ones:

```
# Compute determinant
>>> LA.det(A)

# Compute inverse
>>> LA.inv(A)

# Solve linear system Ax = b
>>> b = np.array([5,6])
>>> x = LA.solve(A, b)
>>> x
array([-4. ,  4.5])

# Verify solution
>>> A @ x
array([5., 6.])
```

We can also compute eigenvalues and eigenvectors:

```
>>> eigenvals, eigenvecs = LA.eig(A)
>>> eigenvals
array([-0.37228132+0.j,  5.37228132+0.j])

>>> eigenvecs
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

26.18 Vector and matrix norms

The **numpy.linalg.norm** function computes various vector and matrix norms. For vectors, the most common norms are: - L_1 norm (Manhattan distance): sum of absolute values - L_2 norm (Euclidean distance): square root of sum of squares - Max norm L_∞: maximum absolute value

[norm]

```
>>> v = np.array([1, -2, 3])
>>> LA.norm(v, ord=1)    # L1 norm
6.0
>>> LA.norm(v)           # L2 norm (default)
3.7416573867739413
>>> LA.norm(v, ord=np.inf)  # Max norm
3.0
```

For matrices, we can compute the Frobenius norm (default) and various matrix norms:

```
>>> A = np.array([[1,2], [3,4]])
>>> LA.norm(A)           # Frobenius norm
5.477225575051661
>>> LA.norm(A, ord=1)    # Maximum column sum
6.0
>>> LA.norm(A, ord=np.inf)  # Maximum row sum
7.0
```

These norms are useful in many applications, such as measuring distances between vectors, checking convergence of numerical algorithms, and analyzing error bounds.

For more information about NumPy's linear algebra capabilities, refer to the official documentation at: https://numpy.org/doc/stable/reference/routines.linalg.html

Chapter 27. Matplotlib

Matplotlib is the most popular package for producing figures: line charts, bar charts, scatter plots, etc. Let's run **ipython**, import the **matplotlib.pyplot** package, and [pyplot, ion] ask it to run in *interactive mode*:

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
<matplotlib.pyplot._IonContext at 0x7fb8c0544670>
```

Matplotlib's default is non-interactive, which means that a figure is displayed only after **plt.show()** is called. Calling **plt.ion()** asks matplotlib to display/update a figure after every operation. This is preferable for experimentation and data science work.

27.1 Your first figure

[plt.plot] Let's plot the function $\sin(x)$ in the range $[0, 2\pi]$:

```
>>> import numpy as np
>>> X = np.linspace(0, 2*np.pi, 20)
>>> plt.plot(X, np.sin(X))
[<matplotlib.lines.Line2D at 0x7fb8401b9730>]
```

A window just like Figure 27.1 should open. Matplotlib automatically chooses the range [plt.close] of the axes, the x/y tick marks and uses the default blue line plot. To close this plot, you can call **plt.close()** or simply close the window. Let's add a plot of $cos(x)$ on top of the current plot. This is done simply by calling **plt.plot** again:

```
>>> plt.plot(X, np.cos(X))
[<matplotlib.lines.Line2D at 0x7fb8902b4e50>]
```

The output is shown in Figure 27.2.

27.2 Opening and closing figures

[plt.figure] By default, all the functions in matplotlib.pyplot draw on top of the last figure. If you want to start a new blank figure, just call **plt.figure()**. Subsequent calls to **plt.plot** will draw onto this figure. To close the last figure call **plt.close()**. To close *all* figures, call **plt.close('all')**. If you close all the figures, every call to a plotting function also creates a new figure.

27.3 Customizations

Matplotlib automatically picks the range of the X/Y axes, the position of the tick marks, the fonts and the colors. However, almost everything is customizable.

[plt.xlim, plt.legend, plt.grid, plt.title, plt.xlabel] Let's see a more complicated example:

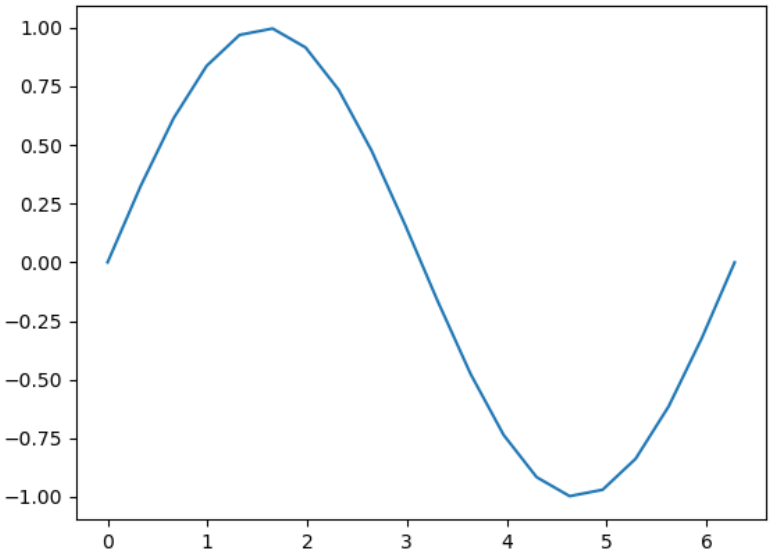


Figure 27.1

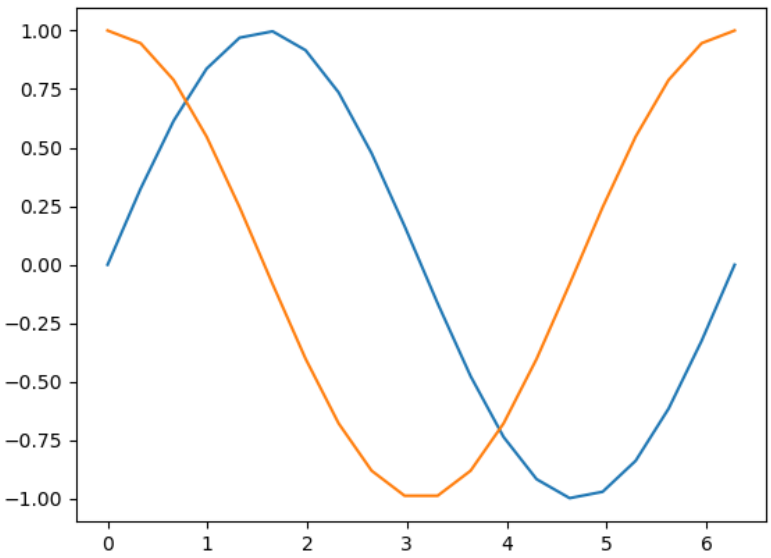


Figure 27.2

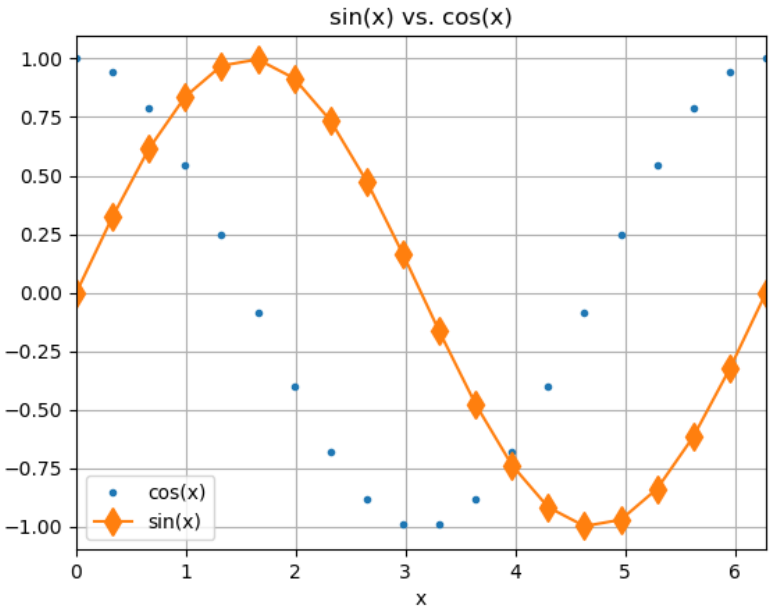


Figure 27.3

```
>>> plt.close('all') # Close all figures
>>> plt.plot(X, np.cos(X), '.', label='cos(x)')
[<matplotlib.lines.Line2D at 0x7fb8d30b9af0>]
>>> plt.plot(X, np.sin(X), 'd-', label='sin(x)', markersize=10) # Big diamond
[<matplotlib.lines.Line2D at 0x7fb8d319dd30>]
>>> plt.xlim([0,2*np.pi]) # Adjust X range
(0.0, 6.283185307179586)
>>> plt.legend() # Plot a legend using the label arguments given above
<matplotlib.legend.Legend at 0x7fb8d3198be0>
>>> plt.grid()
>>> plt.title('sin(x) vs. cos(x)')
Text(0.5, 1.0, 'sin(x) vs. cos(x)')
>>> plt.xlabel('x')
Text(0.5, 47.04444444444444, 'x')
```

Some comments about this example:

- The third argument to **`plt.plot`** is the format string that controls the marker type, line style and also the color. For example **`'d-'`** draws **d**iamond markers connected by lines using the default color. Picking **`'go--'`** would draw big **g**reen circles connected by dashed green lines.
- The function **`plt.legend`** places a legend. To use it you need to specify the labels of the various plots by using a **`label=...`** argument to the **`plt.plot`** function. By default, the

See Figure 27.3 for the results.

27.4 Saving

`plt.savefig('tmp.png')` vs. **`plt.savefig('tmp.pdf')`**

27.5 Scatter plots

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
<contextlib.ExitStack at 0x11b0b2c60>
>>> from numpy.random import multivariate_normal
>>> X = multivariate_normal([0,0], [[1,0.5],[0.5,1]], size=1000)
>>> X.shape
(1000, 2)
>>> plt.plot(X[:,0], X[:,1], '.')
[<matplotlib.lines.Line2D at 0x1215b4560>]
```

27.6 imshow

TODO

```
import numpy as np
np.mgrid[0:5, 0:5]
array([[[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
np.mgrid[-1:1:5j]
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

For more on Matplotlib, refer to the official documentation (in particular, to the example figures) at https://matplotlib.org/

Chapter 28. Randomness

28.1 numpy.random

[numpy.random] The **numpy.random** module can be used to generate random arrays.

```
>>> import numpy.random
>>> numpy.random.seed(7)
>>> numpy.random.random((2,3))
array([[0.07630829, 0.77991879, 0.43840923],
       [0.72346518, 0.97798951, 0.53849587]])
```

To draw numbers uniformly in the range [100,110] we can simply generate U[0, 1] numbers, multiply them by 10 and add 100.

```
>>> import numpy.random as npr
>>> npr.random((2,3))*10 + 100
array([[107.51150339, 102.43939928, 101.06868605],
       [101.62515101, 109.30959779, 101.57511198]])
```

A matrix of independent Gaussian random variables can be drawn by calling the function **numpy.random.normal(mu, sigma, size)** where **mu** is the expectation and **sigma** is the standard deviation. There are many more functions in **numpy.random**. Refer to the documentation or use **help()** to learn more.

```
>>> npr.normal(0, 1, (2,3))
array([[-1.12046301, -0.10200435,  0.10365954],
       [-0.86779016,  0.2103048 , -0.3260374 ]])
```

28.2 Central limit theorem demo using matplotlib

What is the distribution of the sum of 100 fair dice draws? Let's experiment!

```
>>> rpr.randint(1, 7, size=100)
array([2, 6, 3, 1, 5, 3, 5, 1, 3, 4, 3, 5, 2, 6, 3, 2, 6, 2, 5, 2, 1, 5,
       6, 2, 6, 5, 4, 5, 2, 4, 1, 2, 6, 6, 4, 5, 1, 4, 2, 5, 5, 6, 5, 6,
       3, 2, 4, 4, 3, 5, 1, 6, 4, 6, 1, 6, 2, 6, 1, 5, 4, 6, 2, 3, 1, 3,
       1, 3, 2, 2, 6, 6, 4, 2, 1, 1, 2, 2, 2, 6, 4, 3, 1, 6, 3, 3, 5, 3,
       4, 5, 1, 2, 5, 2, 5, 4, 3, 5, 2, 1])
>>> _.sum()
352
```

The mean of $\{1, 2, 3, 4, 5, 6\}$ is 3.5 so by the law of large numbers we expect to get around $3.5 \times 100 = 350$. We can repeat this experiment 1000 times using a nested list comprehension:

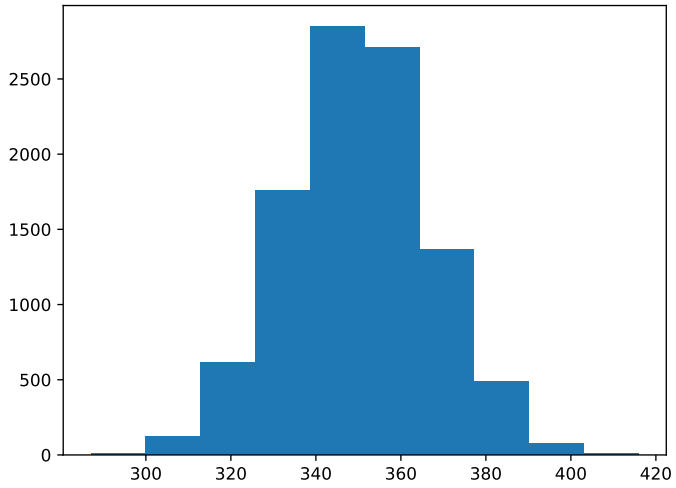
```
>>> results = [npr.randint(1,7,size=1000).sum() for _ in range(1000)]
>>> results[:10]
[3535, 3489, 3455, 3444, 3529, 3539, 3556, 3458, 3429, 3449]
```

[matplotlib]

[histogram plot]

We would like to visualize this with a histogram plot. For this, we will demonstrate the use of the **matplotlib** module:

```
>>> import matplotlib.pyplot as plt
>>> plt.ion() # Interactive mode ON
<contextlib.ExitStack at 0x143d1ae50>
>>> plt.hist(results)
(array([  10.,  123.,  612., 1762., 2846., 2709., 1368.,  486.,   75., 9.]),
 array([287. , 299.9, 312.8, 325.7, 338.6, 351.5, 364.4, 377.3, 390.2,
    403.1, 416. ]), <BarContainer object of 10 artists>)
```



The **plt.ion()** turns on interactive mode. This makes **matplotlib** update the figure after every command. This is what you typically want when doing data science in the interpreter. The default (non-interactive) mode only produces the plot when **plt.show** or **plt.savefig** are called. It is better suited to programs that save figures to files. The **plt.hist** command automatically divides the data into equally-sized bins and draws a bar-plot where the height of each bar is the number of data points in it.

We can tweak the histogram by parameterizing **matplotlib.pyplot.hist**:

```
>>> plt.hist(results, range=(300,400), bins=20)
(array([  29.,   52.,  102.,  178.,  314.,  487.,  695.,  865., 1037.,
       1120., 1157., 1083.,  923.,  700.,  513.,  327.,  189.,  106.,
         58.,   34.]),
 array([300., 305., 310., 315., 320., 325., 330., 335., 340., 345., 350.,
       355., 360., 365., 370., 375., 380., 385., 390., 395., 400.]),
 <BarContainer object of 20 artists>)
```

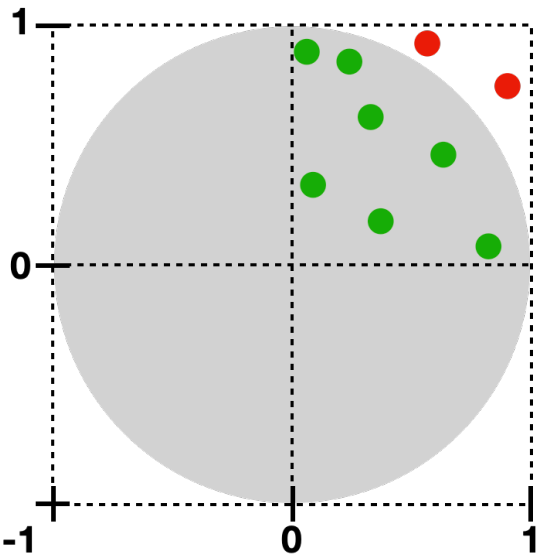
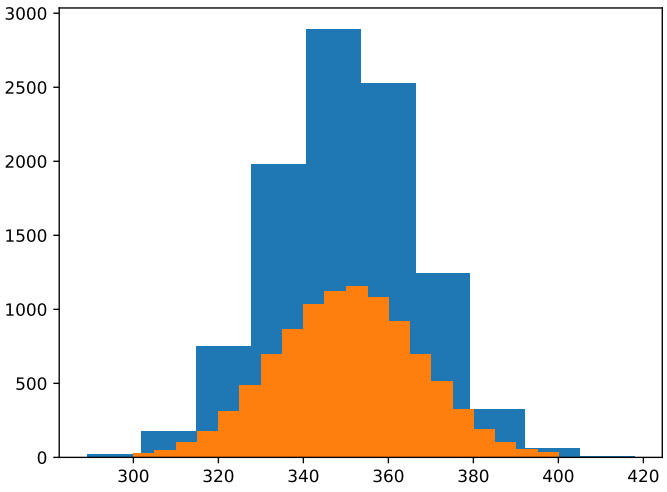


Figure 28.1: The area of this disk is π. We estimate the area of the top-right quarter disk by generating points uniformly in $[0,1]\times[0,1]$ and checking the percentage of points that lie inside the disk. This converges to the ratio of the grey area to its enclosing box.



Note that **matplotlib**'s default behavior is to draw on top of the previous figure. To close the last figure you can use **matplotlib.pyplot.close()** and to close all the figures **matplotlib.pyplot.close('all')**.

28.3 Monte Carlo estimation of π

Computational methods based on random numbers are called *Monte Carlo* methods, after the famous Monte Carlo casino in Monaco. One example is area estimation. Recall the formula for the area of a disk: πr^2. A simple consequence is that by measuring the area of the disk enclosed by the unit circle we get π. We will estimate this by randomly drawing points in the box $[0,1]\times[0,1]$ and checking how many of them have radius ≤ 1.

```
>>> def monte_carlo_pi(n):
...     n_inside = 0
...     for i in range(n):
...         x = numpy.random.random()
...         y = numpy.random.random()
...         n_inside += ((x ** 2 + y ** 2) < 1.0)
...     return 4.0 * n_inside / n
...
>>> monte_carlo_pi(1000)
3.088
>>> monte_carlo_pi(10000)
3.1508
```

Again, we can do this much faster using **numpy**. This will let us use larger values of **n**, thus giving more accurate results.

```
>>> n = 100000000
>>> 4*np.mean(np.sum(numpy.random.random(size=(n,2))**2, axis=1) < 1)
3.14160752
```

Not bad! For computing π there are very fast deteministic methods so randomness is not needed. However, there are many problems for which the deterministic solutions are either too slow or too complicated to use and Monte Carlo methods provide a great alternative.

28.4 The birthday "paradox"

A well-known phenomenon is that in a regular classroom of say 30 students you can often find two that share the same birthday. This is despite the fact that 30 is much smaller than 365. To understand this better, let's ask the following question: in a class of n, what is the probability that there will be at least two students who celebrate their birthday on the same day and month?

To answer this we will code a Monte Carlo simulation:

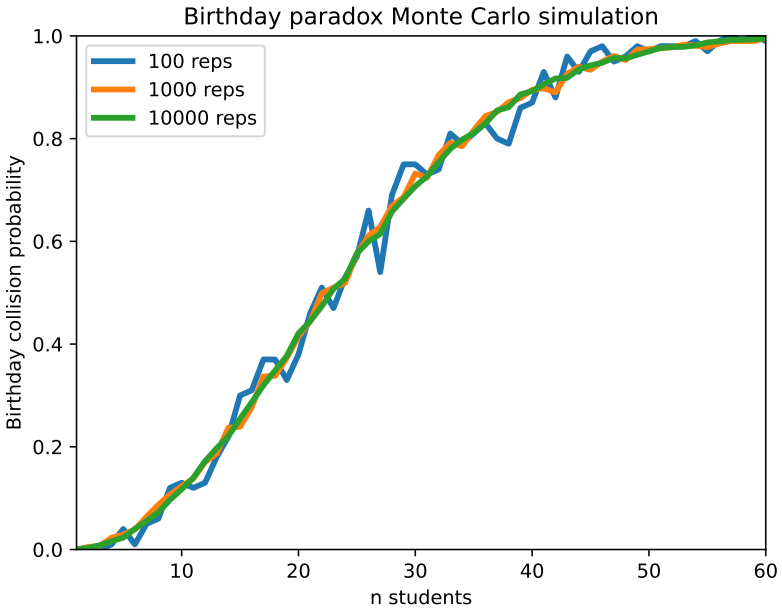
1. Generate a random birthday for each student in the class. We will assume for simplicity that all birthdays are equally likely.
2. Check if there are two students with the same birthday.
3. Repeat the above many times and calculate the percentage of classes that have a birthday collision.

By repeating the experiment many times the proportion of classes with a birthday collision should converge (by the law of large numbers) to the probability of a collision. To draw a random birthday for each student we can **numpy.random.randint** function:

```
>>> n = 30
>>> bdays = npr.randint(0,365,size=n)
>>> bdays
array([241, 139,  96,  75, 159,  32, 342,  52, 124,  97,  43, 271,  24,
       178, 226,  70, 141, 135,  30, 304,  45,  20,  43, 179,  80, 314,
       169,  81,  67, 216])
```

How do we check for a collision? The easiest way is to check if the number of unique elements in **bdays** is equal to **n**.

```
>>> len(set(bdays))
29
```



This means that there is one collision. Indeed, 43 is repeated twice. Repeating this experiment is easy:

```
>>> [len(set(npr.randint(0,365,size=n))) < n for _ in range(100)]
[True,
False,
False,
True,
True,
False,
True,
.
.
.
]
```

All that is left is to calculate the proportion of **True** values in the list above. This can be done easily by converting the list to a **numpy** array and using the **mean** function.

```
>>> np.array([len(set(npr.randint(0,365,size=n))) <  for _ in
    range(100)]).mean()
0.66
```

Let's put it all together:

```
>>> def birthday_collision_prob(n, reps):
...     unique_bdays = np.array([len(set(npr.randint(0,365,size=n))) for _
    in range(reps)])
...     collision_mask = unique_bdays < n
...     return collision_mask.mean()
...
>>> birthday_collision_prob(60,1000)
0.991
>>> X = np.arange(1,61)
>>> Y = [birthday_collision_prob(n, 100) for n in X]
>>> plt.plot(X, Y, linewidth=3, label='100 reps')
[<matplotlib.lines.Line2D at 0x16b74e9d0>]
```

```
>>> plt.close()
>>> X = np.arange(1,61)
>>> Y = [birthday_collision_prob(n, 100) for n in X]
>>> plt.plot(X, Y, linewidth=3, label='100 reps')
[<matplotlib.lines.Line2D at 0x169cb9bd0>]
>>> Y = [birthday_collision_prob(n, 1000) for n in X]
>>> plt.plot(X, Y, linewidth=3, label='1000 reps')
[<matplotlib.lines.Line2D at 0x168cbcc50>]
>>> Y = [birthday_collision_prob(n, 10000) for n in X]
>>> plt.plot(X, Y, linewidth=3, label='10000 reps')
[<matplotlib.lines.Line2D at 0x168d4f4d0>]
>>> plt.xlim([1,60])
(1.0, 60.0)
>>> plt.xlabel('n students')
Text(0.5, 47.04444444444444, 'n students')
>>> plt.ylabel('Birthday collision probability')
Text(85.31944444444443, 0.5, 'Birthday collision probability')
>>> plt.title('Birthday paradox Monte Carlo simulation')
Text(0.5, 1.0, 'Birthday paradox Monte Carlo simulation')
>>> plt.ylim([0,1])
(0.0, 1.0)
>>> plt.legend()
>>> plt.plt.savefig('birthday-paradox-figure.pdf')
```

28.5 Application: recurrency of random walks

TODO: explain Random walkers in one and two dimensions always return to the origin. (this is not true for higher dimensions)

```
>>> np.mean([0 in ((np.random.randint(0,2,size=100000)*2-1).cumsum()) for
    rep in range(100)])
1.0
```

```
>>> Z = (np.random.randint(0,2,size=(10,2))*2-1).cumsum(axis=0)
>>> ((Z == [0,0]).sum(axis=1) == 2).any()
True
>>> np.mean([(((np.random.randint(0,2,size=(10000,2))*2-1).cumsum(axis=0) ==
    [0,0]).sum(axis=1) == 2).any() for reps in range(100)])
0.66
```

Chapter 29. Trees

A binary search tree (BST) is a binary tree where each node has a key, and satisfies the following recursive invariant:

- The left and right subtrees are either None or a binary search trees.
- The key of each node in the left subtree is smaller than or equal to the key of the node itself.
- The key of each node is right subtree is greater than the key of the node itself.

Here is a simple implementation of a binary search tree in Python. The 'BST' class has a constructor that takes a key, optional left and right subtrees, and optional data. It also has methods to find the minimum node in the tree and to return the nodes in in-order traversal.

```
class BST:
    def __init__(self, key, left=None, right=None, data=None):
        if left is not None:
            assert left.key <= key
        if right is not None:
            assert right.key > key
        self.key = key
        self.left = left
        self.right = right
        self.data = data

    def __repr__(self):
        return f'BST({repr(self.key)}, {repr(self.left)},
    {repr(self.right)}, {repr(self.data)})'

    def min_node(self):
        return self.left.min_node() if self.left else self

    def in_order(self):
        left_nodes_in_order = self.left.in_order() if self.left else []
        right_nodes_in_order = self.right.in_order() if self.right else []
        return left_nodes_in_order + [self] + right_nodes_in_order
```

This allows us to create trees like this:

```
tree = BST(6,
           BST(1,
               BST(1),
               BST(6,
                   BST(3),
                   None)),
           BST(7,
               None,
               BST(8,
```

```
                BST(8),
                BST(9))))
```

To make these objects nicer to manipulate in the interpreter, we can add a **__repr__** method that returns a string representation of the object.

```
    def __repr__(self):
        return f'BST({repr(self.key)}, {repr(self.left)},
    {repr(self.right)}, {repr(self.data)})'

    def min_node(self):
        return self.left.min_node() if self.left else self

    def in_order(self):
        left_nodes_in_order = self.left.in_order() if self.left else []
        right_nodes_in_order = self.right.in_order() if self.right else []
        return left_nodes_in_order + [self] + right_nodes_in_order
```

Exercise 29.0.1. Write a method that finds the node with the minimum node in a tree. Since the left subtree of a binary search tree contains only nodes with keys less than or equal to the key of the root node, we will write a method that recursively goes to the left subtree, until we reach a node with no left child.

```
    def min_node(self):
        return self.left.min_node() if self.left else self
```

Exercise 29.0.2. Write a method that returns all of the nodes in a tree ordered by their keys. By the tree ordering property we need to recursively traverse the left subtree (in order) then take the current node, and finally traverse the right subtree.

```
    def in_order(self):
        left_nodes_in_order = self.left.in_order() if self.left else []
        right_nodes_in_order = self.right.in_order() if self.right else []
        return left_nodes_in_order + [self] + right_nodes_in_order
```

Chapter 30. Pandas

Pandas is a popular Python library for working with tabular data. Let's import it:

```
>>> import pandas as pd
```

Pandas contains two main classes for storing and manipulating data:

- **pandas.Series** for typed arrays, similar to a one-dimensional **numpy.array**.
- **pandas.DataFrame** for storing tables, where each column is a **pandas.Series**. However, unlike two-dimensional NumPy arrays where all elements have the same **dtype**, in a DataFrame, each column can have a different **dtype**.

30.1 The pandas.Index class

[pandas.Index]

Both **Series** and **DataFrame** objects have an associated **Index** object, which holds the row labels.

```
>>> idx_abc = pd.Index(['a', 'b', 'c'])
>>> idx_321 = pd.Index([3,2,1,0])
>>> idx_abc
Index(['a', 'b', 'c'], dtype='object')
>>> idx_321
Index([3, 2, 1, 0], dtype='int64')
>>> len(idx_abc)
3
>>> idx_abc[1]
'b'
```

Unlike Python lists and NumPy arrays, Index objects are *immutable*:

```
>>> idx_abc[1] = 'zzz'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 idx_abc[1] = 'zzz'

File
    ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:5157,
    in Index.__setitem__(self, key, value)
   5155 @final
   5156 def __setitem__(self, key, value):
-> 5157     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

Question. Why are Index objects immutable?
Answer. So that they could be shared between different data structures without the risk of one of them modifying the index in a way that would affect the other.

30.2 The pandas.Series class

[pandas.Series] A pandas **Series** holds a sequence of labeled values:

```
>>> s = pd.Series([97, 98, 99], idx_abc)
>>> s
a    97
b    98
c    99
dtype: int64
```

The indices a,b,c are the labels and the values are 97,98,99. Similar to NumPy arrays, the dtype is inferred from the input list:

```
>>> s = pd.Series([97, 98, 99.999], idx_abc)
>>> s
a    97.000
b    98.000
c    99.999
dtype: float64
```

You don't have to construct an **Index** object before initializing a new **Series** object. The labels can be given as the second argument to **pandas.Series**:

```
>>> s = pd.Series([97, 98, 99], ['a', 'b', 'c'])
```

Another way to create a Series is from a dictionary:

```
>>> pd.Series({'a': 97, 'b': 98, 'c': 99})
a    97
b    98
c    99
dtype: int64
>>> s.to_dict()    # Convert back to dict
{'a': 1, 'b': 2, 'c': 3}
```

30.2.1 Indexing with iloc

Series objects can be indexed by integers, just like Python lists and NumPy arrays using the **.iloc** method: [iloc]

```
    >>> s = pd.Series(range(100,200,10), list('abcdefghij'))
    >>> s
    a    100
    b    110
    c    120
    d    130
    e    140
    f    150
    g    160
    h    170
    i    180
    j    190
    dtype: int64
    >>> s.iloc[1]
    110
    >>> s.iloc[-1]
    190
    >>> s.iloc[1] = 1234
    >>> s
    a     100
    b    1234
```

```
c     120
d     130
e     140
f     150
g     160
h     170
i     180
j     190
dtype: int64
```

Slices are also supported. Just like NumPy arrays, they are *views* of the Series (not copies). So modifying a slice changes the original Series.

```
>>> s1_3 = s.iloc[1:3]
>>> s1_3
b    1234
c     120
dtype: int64
>>> s1_3[1] = 120120
>>> s
a       100
b      1234
c    120120
d       130
e       140
f       150
g       160
h       170
i       180
j       190
dtype: int64
```

You can even use fancy indexing like in NumPy arrays:

```
>>> s[[7,9]]
h    170
j    190
dtype: int64
```

30.2.2 Indexing with loc

The **pandas.Series.loc** method supports label-based indexing (like Python dictionaries):

[loc]

```
>>> s.loc['d']
130
>>> s.loc['d'] = 0
>>> s
a       100
b      1234
c    120120
d         0
e       140
f       150
g       160
h       170
i       180
j       190
dtype: int64
```

Slicing is supported by **.loc**, but it has a quirk:

```
>>> s.loc['d'] = 999
>>> s.loc['g':'i']
g    160
h    170
i    180
dtype: int64
>>> s.loc['g':'i'] = 0
>>> s
a       100
b      1234
c    120120
d       999
e       140
f       150
g         0
h         0
i         0
j       190
dtype: int64
```

Note the these slices include the end label of the slice! This is different from lists and NumPy arrays.

30.2.3 Indexing with [ ]

You can index **pandas.Series** using the **[]** operator. However, it behaves either like **.loc** or **.iloc** depending on the context:

```
>>> s = pd.Series([100,200,300], [2,4,6])
>>> s[2]
100
>>> s = pd.Series([100,200,300], ['a','b','c'])
>>> s[2]
300
```

Because of this ambiguous behavior, its probably best to stick with **loc** and **iloc**.

30.2.4 Vectorized Series operations

Like NumPy arrays, Series support vectorized operations:

```
>>> s = pd.Series(range(100,150,10), list('abcde'))
>>> s
a    100
b    110
c    120
d    130
e    140
dtype: int64
>>> s*2
a    200
b    220
c    240
d    260
e    280
dtype: int64
>>> np.sqrt(s)
a    10.000000
b    10.488088
c    10.954451
d    11.401754
```

```
e    11.832160
dtype: float64
```

Just like NumPy arrays, you can create boolean masks and index using them:

```
>>> mask = s > 120
>>> mask
a    False
b    False
c    False
d     True
e     True
dtype: bool
>>> s[mask]
d    130
e    140
dtype: int64
>>> s[(s > 110) & (s < 140)] # Must use "&", not "and"
c    120
d    130
dtype: int64
```

30.2.5 Binary operations between pandas.Series

Let's add two **Series** objects:

```
>>> sa = pd.Series([100,200,300], ['a','b','c'])
>>> sb = pd.Series([1,2,3,4], ['a','c','cat','meeow'])
>>> sa+sb
a        101.0
b          NaN
c        302.0
cat        NaN
meeow      NaN
dtype: float64
```

Binary operations on **Series** objects respect the row labels. Tha **NaN** values are short for *Not a Number*. This is the result of adding a number to a missing value.

30.3 DataFrames

A **pandas.DataFrame** is a table object, where the columns are **Series** objects that all share the same **Index**.

30.3.1 Creating DataFrames

Let's create a table of the three densest cities. First we define the columns as **Series** objects:

```
>>> cities = ['Giza', 'Manila', 'Mandaluyong']
>>> pop = pd.Series([4_432_915, 1_846_600, 425_758], cities)
>>> area = pd.Series([98,43,11], cities)
```

Then we create the **DataFrame** object:

```
>>> df = pd.DataFrame({'population': pop, 'area [km^2]': area})
>>> df
            population  area [km^2]
Giza           4432915           98
Manila         1846600           43
Mandaluyong     425758           11
```

The index (row labels) of the data frame is the same as the index of the **Series** objects. An altrnative way to create a **DataFrame** is to use a list of dictionaries:

```
>>> pd.DataFrame({'population': [4432915, 1846600, 425758],
                  'area [km^2]': [98, 43, 11]},
                  index=['Giza', 'Manila', 'Mandaluyong'])
               population  area [km^2]
Giza              4432915           98
Manila            1846600           43
Mandaluyong        425758           11
```

And yet another way is from a list of lists:

```
>>> data = [[4432915, 98],
...         [1846600, 43],
...         [425758, 11]]
...
>>> df = pd.DataFrame(data, columns=['population', 'area [km^2]'],
    index=['Giza', 'Manila', 'Mandaluyong'])
```

30.3.2 Reading and writing columns

You can retrieve the columns of a **DataFrame** using dictionary-style access:

```
>>> df['area [km^2]']
Giza           98
Manila         43
Mandaluyong    11
Name: area [km^2], dtype: int64

>>> type(_)
pandas.core.series.Series
```

We see that a column is just a **pandas.Series** object. For columns with names that are valid Python identifiers, you can also access them as attributes: (this doesn't work for **area [km2̂]** because of the space in the name)

```
>>> df.population
Giza           4432915
Manila         1846600
Mandaluyong     425758
Name: population, dtype: int64
```

New columns can be added to a **DataFrame**, just like adding new items to a dictionary. Let's add a new column for the population density:

```
>>> df['density'] = df['population']/df['area [km^2]']
>>> df
               population  area [km^2]       density
Giza              4432915           98  45233.826531
Manila            1846600           43  42944.186047
Mandaluyong        425758           11  38705.272727
```

If we don't like the new column, we can remove it using the **del** keyword:

```
>>> del df['density']
>>> df
               population  area [km^2]
Giza              4432915           98
Manila            1846600           43
Mandaluyong        425758           11
```

30.3.3 Accessing rows/columns with iloc

```
>>> df.iloc[1] # Fetch the second row
population     1846600
area [km^2]         43
Name: Manila, dtype: int64
>>> type(_)
pandas.core.series.Series

>>> df.iloc[:,1] # Accessing the second column
Giza           98
Manila         43
Mandaluyong    11
Name: area [km^2], dtype: int64
>>> type(_)
pandas.core.series.Series

>>> df.iloc[:2,:] # Accessing the first two rows
        population  area [km^2]
Giza       4432915           98
Manila     1846600           43
>>> type(_)
pandas.core.frame.DataFrame
```

Similarly, **`.loc`** can be used to access rows and columns by name:

```
>>> df.loc['Giza':'Manila', 'population']
Giza      4432915
Manila    1846600
Name: population, dtype: int64
```

30.3.4 Boolean indexing

```
>>> df.population > 1000000
Giza            True
Manila          True
Mandaluyong    False
Name: population, dtype: bool

>>> df[df.population > 1000000]
        population  area [km^2]
Giza       4432915           98
Manila     1846600           43
```

30.4 Reading DataFrames

30.4.1 Iterating over the rows of a DataFrame

Let's read an excel sheet Pandas can read tables from files such as CSV, Excel, etc. Let's read an Excel sheet:

```
>>> df = pd.read_excel('trip-expenses.xlsx')
>>> df
        Date      Who       $                                  What
0 2025-01-01   Joseph   42.50                        Emergency pizza
1 2025-01-01   Joseph   42.88  Matching tropical shirts for everyone
2 2025-01-02   Yoseph   24.99                      Unicorn pool float
3 2025-01-02   Yoseph   38.90                        Flip-flop rental
4 2025-01-03  Nehorai   89.75            Wrong train tickets (oops!)
5 2025-01-03  Nehorai   47.50                           Train tickets
6 2025-01-04   Yoseph   35.00                            Palm reading
```

```
7 2025-01-04  Nehorai   48.50                    Cucumber face masks
8 2025-01-04   Joseph  150.00                              Goat ride
```

30.4.2 Iterating over DataFrames

You can iterate over the rows as follows:

```
>>> for (index,row) in df.iterrows():
...     print(index)
...     print('----')
...     print(row)
...     print('====')
...
0
----
Date    2025-01-01 00:00:00
Who                  Joseph
$                      42.5
What         Emergency pizza
Name: 0, dtype: object
====
1
----
Date                       2025-01-01 00:00:00
Who                                    Joseph
$                                       42.88
What    Matching tropical shirts for everyone
Name: 1, dtype: object
====
2
----
Date    2025-01-02 00:00:00
Who                  Yoseph
$                     24.99
What     Unicorn pool float
Name: 2, dtype: object
====.
.
.
.
```

Each row is a **pandas.Series** object:

```
>>> row
Date    2025-01-04 00:00:00
Who                  Joseph
$                     150.0
What              Goat ride
Name: 8, dtype: object
>>> row['$']
150.0
```

Another way to use **itertuples**:

```
>>> for t in df.itertuples():
...     print(t)
...
Pandas(Index=0, Date=Timestamp('2025-01-01 00:00:00'), Who='Joseph',
    _3=42.5, What='Emergency pizza')
Pandas(Index=1, Date=Timestamp('2025-01-01 00:00:00'), Who='Joseph',
    _3=42.88, What='Matching tropical shirts for everyone')
```

```
Pandas(Index=2, Date=Timestamp('2025-01-02 00:00:00'), Who='Yoseph',
    _3=24.99, What='Unicorn pool float')
Pandas(Index=3, Date=Timestamp('2025-01-02 00:00:00'), Who='Yoseph',
    _3=38.9, What=' Flip-flop rental')
Pandas(Index=4, Date=Timestamp('2025-01-03 00:00:00'), Who='Nehorai',
    _3=89.75, What='Wrong train tickets (oops!)')
Pandas(Index=5, Date=Timestamp('2025-01-03 00:00:00'), Who='Nehorai',
    _3=47.5, What='Train tickets')
Pandas(Index=6, Date=Timestamp('2025-01-04 00:00:00'), Who='Yoseph',
    _3=35.0, What='Palm reading')
Pandas(Index=7, Date=Timestamp('2025-01-04 00:00:00'), Who='Nehorai',
    _3=48.5, What='Cucumber face masks')
Pandas(Index=8, Date=Timestamp('2025-01-04 00:00:00'), Who='Joseph',
    _3=150.0, What='Goat ride')
```

These objects behave like tuples with names for the alphanumeric fields. You can access the fields by position or by name:

```
>>> t[4]
'Goat ride'
>>> t.What
'Goat ride'
```

The **itertuples** method can also be combined with tuple unpacking:

```
>>> for (index, date, who, cost, what) in df.itertuples():
...     print(date)
...
2025-01-01 00:00:00
2025-01-01 00:00:00
2025-01-02 00:00:00
2025-01-02 00:00:00
2025-01-03 00:00:00
2025-01-03 00:00:00
2025-01-04 00:00:00
2025-01-04 00:00:00
2025-01-04 00:00:00
```

Iterating over the rows using **iterrows** and **itertuples** is often a quick way to get things done but not the fastest way to process DataFrames.

30.4.3 Grouping and aggregating

```
>>> df = pd.read_excel('trip-expenses.xlsx')
>>> df
        Date      Who       $                                  What
0 2025-01-01   Joseph   42.50                      Emergency pizza
1 2025-01-01   Joseph   42.88  Matching tropical shirts for everyone
2 2025-01-02   Yoseph   24.99                  Unicorn pool float
3 2025-01-02   Yoseph   38.90                    Flip-flop rental
4 2025-01-03  Nehorai   89.75        Wrong train tickets (oops!)
5 2025-01-03  Nehorai   47.50                       Train tickets
6 2025-01-04   Yoseph   35.00                      Palm reading
7 2025-01-04  Nehorai   48.50                Cucumber face masks
8 2025-01-04   Joseph  150.00                          Goat ride
>>> df.groupby('Who')['$'].sum()
Who
Joseph     235.38
Nehorai    185.75
Yoseph      98.89
Name: $, dtype: float64
```

30.5 Scraping tables from the web

```
>>> out =
    pd.read_html('https://en.wikipedia.org/wiki/List_of_non-water_floods')
>>> out[0]
    0                                                  1
0 NaN  This article needs additional citations for ve...
>>> out[1]
                                                 Name  ...
         Location
0                                 London Beer Flood  ...
    London, England
1                                Dublin whiskey fire  ...
    Dublin, Ireland
2                               Great Molasses Flood  ...       Boston,
    Massachusetts, U.S.
3     Rockwood & Company shipping department fire  ...               New
    York City, U.S.
4                                  Aberfan disaster  ...
    Aberfan, Wales, UK
5                   Church Rock uranium mill spill  ...     Church Rock,
    New Mexico, U.S.
6                            Wisconsin butter flood  ...          Madison,
    Wisconsin, U.S.
7   Kingston Fossil Plant coal fly ash slurry spill  ...         Kingston,
    Tennessee, U.S.
8                      Ajka alumina plant accident  ...
    Ajka, Hungary
9                              Mariana dam disaster  ...
    Mariana, Brazil
10                         Pepsi fruit juice flood  ...
    Lebedyan, Russia
11                      Levira Distiller wine flood  ...  Sao Lourenco do
    Bairro, Portugal
[12 rows x 4 columns]
>>> df = out[1]
>>> df.describe()
                        Name             Date Composition of flood
    Location
count                   12               12                    12
     12
unique                  12               12                    12
     12
top     London Beer Flood  17 October 1814                 Beer  London,
    England
freq                     1                1                     1
      1
>>> df.columns
Index(['Name', 'Date', 'Composition of flood', 'Location'], dtype='object')
>>> df.loc[:,['Date', 'Name', 'Composition of flood']]
                    Date                                                Name
                                Composition of flood
0     17 October 1814                                  London Beer Flood
                                           Beer
1        18 June 1875                                   Dublin whiskey fire
                                        Whiskey
2     15 January 1919                                 Great Molasses Flood
                                       Molasses
3         12 May 1919      Rockwood & Company shipping department fire
                     Molten chocolate and butter
```

```
4     21 October 1966                                  Aberfan disaster
                            Mine spoil and water
5        16 July 1979                  Church Rock uranium mill spill
                                  Uranium tailings
6          3 May 1991                           Wisconsin butter flood
              Butter, cheese, and processed meat
7    22 December 2008  Kingston Fossil Plant coal fly ash slurry spill
                 Coal byproducts mixed with water
8      4 October 2010                       Ajka alumina plant accident
    Bauxite residue mixed with water (caustic slud...
9     5 November 2015                             Mariana dam disaster
                       Tailings mixed with water
10      25 April 2017                           Pepsi fruit juice flood
                                    Various juices
11  10 September 2023                      Levira Distiller wine flood
                                   Red wine
>>> df.loc[:,['Date', 'Name']]
                    Date                                                Name
0     17 October 1814                                  London Beer Flood
1        18 June 1875                                 Dublin whiskey fire
2     15 January 1919                                Great Molasses Flood
3         12 May 1919      Rockwood & Company shipping department fire
4     21 October 1966                                   Aberfan disaster
5        16 July 1979                    Church Rock uranium mill spill
6          3 May 1991                             Wisconsin butter flood
7    22 December 2008  Kingston Fossil Plant coal fly ash slurry spill
8      4 October 2010                       Ajka alumina plant accident
9     5 November 2015                             Mariana dam disaster
10      25 April 2017                           Pepsi fruit juice flood
11  10 September 2023                      Levira Distiller wine flood
```

For more on pandas, refer to the official documentation at: https://pandas.pydata.org/docs/

Part IV

Appendices

Appendix A. Visual Studio Code installation

Visual Studio Code, or VSCode for short, is a very popular development environment for Python programmers with many features, including a built-in editor, debugger, LLM completions, etc. If you don't already have a favorite code editor, try it out!

A.1 Installing VSCode

Install it from here:
https://code.visualstudio.com

Windows users can follow this installation video:
https://www.youtube.com/watch?v=cu_ykIfBprI

A.2 Setting up VSCode for Python

Installing the Python extensions.

1. Click on the Extensions button on the left-most panel of VSCode. It looks like 4 blocks stacked in a 2x2 pattern, where the top-right block is at a 45 degree angle.
2. Search for **`Python`** in the search box and find the **`Python`** extension by Microsoft. Click on it and then click Install.

Selecting the correct Python interpreter: Open VSCode, then type **`Ctrl+Shift+P`** on Windows (**`Cmd+Shift+P`** on MacOS) → **`Python: Select interpreter`**. Then select the conda Python interpreter.[1]

Basic VSCode workflow: Suppose you have a working folder named **`intro-class/ps03`** with a file named **`sol03.py`** inside it. To work on this file:

1. Open VSCode.
2. From the menu, click **`File`** → **`Open Folder...`**. Then navigate to **`intro-class`** and choose your working folder **`ps03`** that is inside it.
3. Open the file **`sol03.py`** (or create a new file if it does not exist).
4. Edit the file by writing a function. e.g.

[1] VSCode actually runs a lot of Python code in order to understand your code. This step asks it to use the same version of the Python interpreter that was installed by Anaconda.

```
def myfunc():
    return 'Hello there'
```

5. Save the file using **Ctrl+S** on Windows (**Cmd-S** on MacOS).

6. To test the code you just wrote, open IPython in the Windows Powershell (Terminal on MacOS). Import your code using **from sol03 import *** and run your function by calling **myfunc()**.

7. Alternatively, you can open the shell as sub-window inside VSCode by selecting **Terminal** → **New Terminal** from the menu bar and then running IPython as usual.

Warning! The code that's already loaded in IPython will not change even if you save the file **sol03.py** and re-import it, unless you explicitly reload the module using the **importlib.reload** function. To make IPython automatically auto-reload modified files, follow Appendix B and you'll never have this problem again.

Appendix B. Module reloading madness and IPython's autoreload

Use a text editor to write a file **blah.py** with the following code:[1]

```
answer = 42
print('The answer is', answer)
```

You can run this by typing **python blah.py** in the command line, or you can import this file as a Python module:

```
>>> import blah
The answer is 42
```

When you import this module, the variable **answer** is stored in **blah.answer**

```
>>> dir(blah)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
    '__name__', '__package__', '__spec__', 'answer']
>>> blah.answer
42
```

Change **blah.py** to have **answer = 0** and import it again in the same Python shell:

```
>>> import blah
>>> blah.answer
42
```

What kind of evil is this??? For efficiency reasons, Python will only read a module from disk the first time it is imported! Notice also that nothing is printed the second time because the module is not re-run. We can force Python to reload the module using the **importlib** module:

```
>>> import importlib
>>> importlib.reload(blah)
The answer is 0
<module 'blah' from '/Users/mosco/blah.py'>
>>> blah.answer
0
```

This behavior can be extremely annyoing when you want to go back and forth between updating your code and running it in the Python interpreter. Follow the next section to configure IPython so that it always reloads all modified Python modules from disk.

[1] If you don't have a favorite text editor installed, or don't know what a text editor is, follow the instructions in A to install Visual Studio Code.

B.1 Configuring IPython to use autoreload

Create a file **00.py** with the following lines:

00.py

```
from IPython import get_ipython
ipython = get_ipython()
print("Running %autoreload 2 to reload modules automatically")
ipython.run_line_magic('load_ext', 'autoreload')
ipython.run_line_magic('autoreload', '2')
```

Windows users: Move the file **00.py** you just created to the IPython startup directory: `C:\Users\`**YourUserName**`\.ipython\profile_default\startup` (replace **YourUserName** with your Windows user name)

Mac/Linux users: Move the file **00.py** to the IPython startup folder under your homedir: `~/.ipython/profile_default/startup/`

IPython should now print **Running %autoreload 2 to reload modules automatically** whenever it starts up. Any loaded Python modules should now automatically reload whenever their file is saved.

Appendix C. Streamlit

Streamlit is a Python library for creating data-centric web applications.

Install it with pip:

```
pip install streamlit
```

Create a file **st_hello.py** with the following content:

```
import streamlit as st
st.write('Hello, world!')
```

Run this in the command line:

```
> streamlit run st_hello.py
```

It should open a web browser window that displays the text *Hello, world!*. In the command-line window you will see a message like:

```
  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://10.0.1.9:8501
```

As long as this process is running, you can change your code at **st_hello.py** and reload the web page. You don't have to rerun **streamlit**.

C.1 Plotting with Streamlit

Let's redo the dice example from the Matplotlib chapter, but this time with Streamlit.

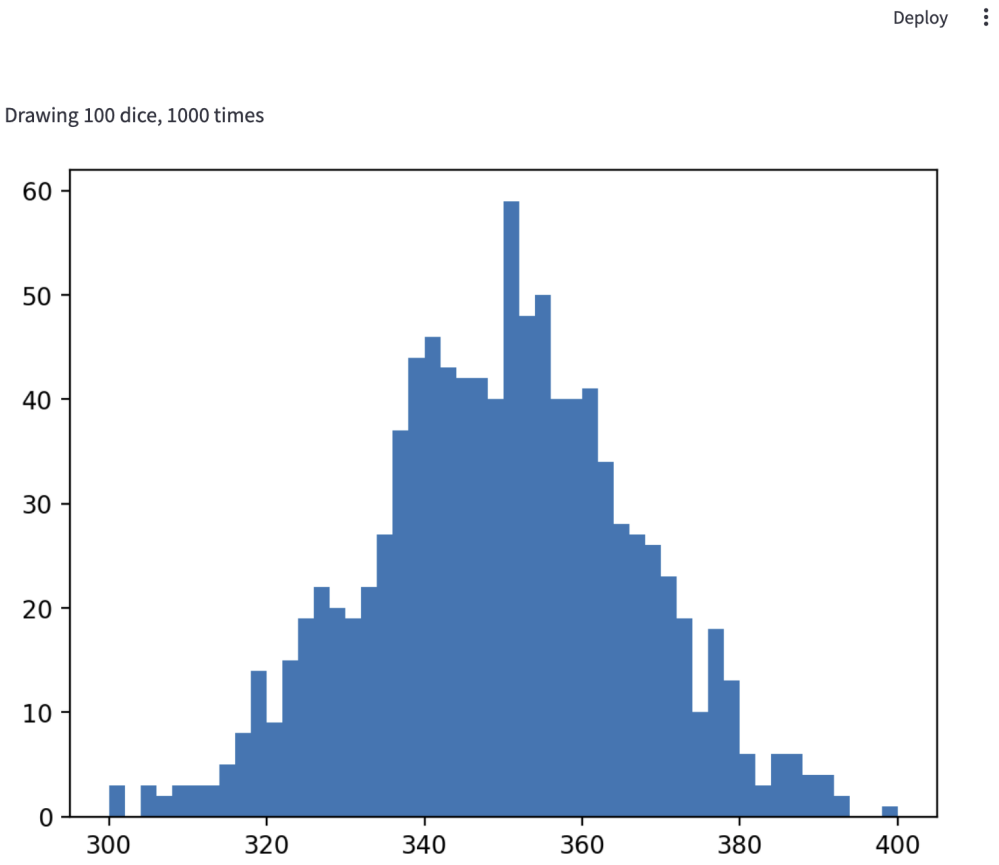
```
import streamlit as st
import numpy as np
import matplotlib.pyplot as plt

NUM_DICE = 100; NUM_REPS = 1000
st.write(f'Drawing {NUM_DICE} dice, {NUM_REPS} times')

results = np.random.randint(1,7,size=(NUM_REPS, NUM_DICE)).sum(axis=1)

fig, ax = plt.subplots()
ax.hist(results, range=(300,400), bins=50)
st.pyplot(fig)
```

Try reloading the streamlit page. You should see something like the following:



C.2 Adding sliders for interactivity

Streamlit supports interactive widgets such as sliders and radio buttons. Let's add a slider to control the number of repetitions.

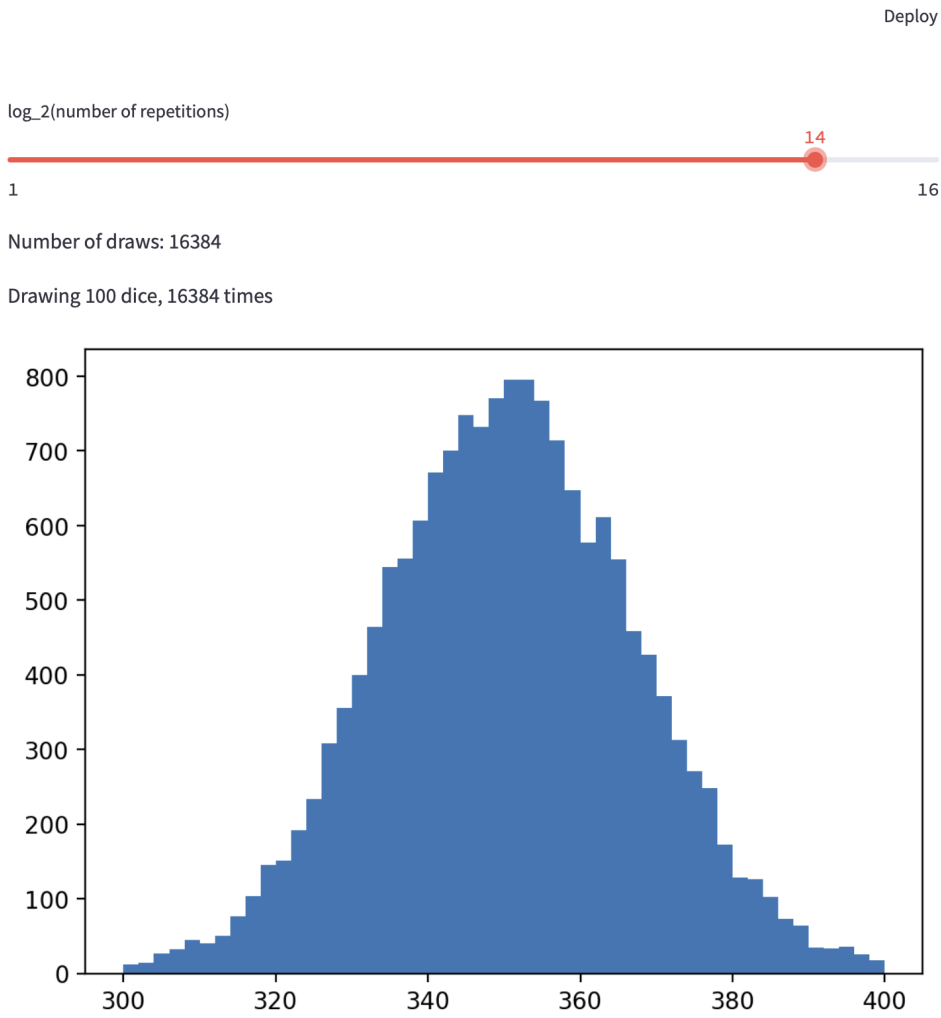
```
import streamlit as st
import numpy as np
import matplotlib.pyplot as plt

log2_NUM_REPS = st.slider('log_2(number of repetitions) ', min_value=1,
    max_value=16, value=1, step=1)
NUM_REPS = 2**log2_NUM_REPS
st.write(f'Number of draws: {NUM_REPS}')

NUM_DICE = 100
st.write(f'Drawing {NUM_DICE} dice, {NUM_REPS} times')

results = np.random.randint(1,7,size=(NUM_REPS, NUM_DICE)).sum(axis=1)

fig, ax = plt.subplots()
ax.hist(results, range=(300,400), bins=50)
st.pyplot(fig)
```



This was just a taste of streamlit. For more, check out the excellent documentation at https://docs.streamlit.io/.

Appendix D. Extra material

D.1 Largest ones block

Slow solution:

```
import numpy as np

def has_ones_block(arr2D, k):
    (n_rows, n_cols) = arr2D.shape
    for i in range(n_rows - k + 1):
        for j in range(n_cols - k + 1):
            square_at_ij = arr2D[i:i+k, j:j+k]
            if np.all(square_at_ij == 1):
                return True
    return False

def largest_ones_block(arr2D):
    (n_rows, n_cols) = arr2D.shape
    return max(k for k in range(min(n_rows, n_cols) + 1) if
    has_ones_block(arr2D, k))
```

Faster solution:

```
def largest_ones_block_fast(arr2D):
  assert arr2D.ndim == 2
  X = arr2D
  k = 0
  while np.any(X == 1) and X.shape[0] > 0 and X.shape[1] > 0:
      k += 1
      X = X[:-1,:-1] & X[:-1,1:] & X[1:,:-1] & X[1:,1:]
  return k

def largest_ones_block_fastest(arr2D):
(n_rows, n_cols) = arr2D.shape
X = arr2D
Z = np.zeros_like(X, dtype=np.int64)
for i in range(1, n_rows):
  for j in range(1, n_cols):
    Z[i,j] = min(Z[i-1,j-1], Z[i-1,j], Z[i,j-1]) + 1 if X[i,j] == 1 else 0
return np.max(Z)
```

Fastest solution:

```
def largest_ones_block_fastest(arr2D):
  (n_rows, n_cols) = arr2D.shape
  X = arr2D
  Z = np.zeros_like(X, dtype=np.int64)
  for i in range(1, n_rows):
    for j in range(1, n_cols):
      Z[i,j] = min(Z[i-1,j-1], Z[i-1,j], Z[i,j-1]) + 1 if X[i,j] == 1 else 0
  return np.max(Z)
```