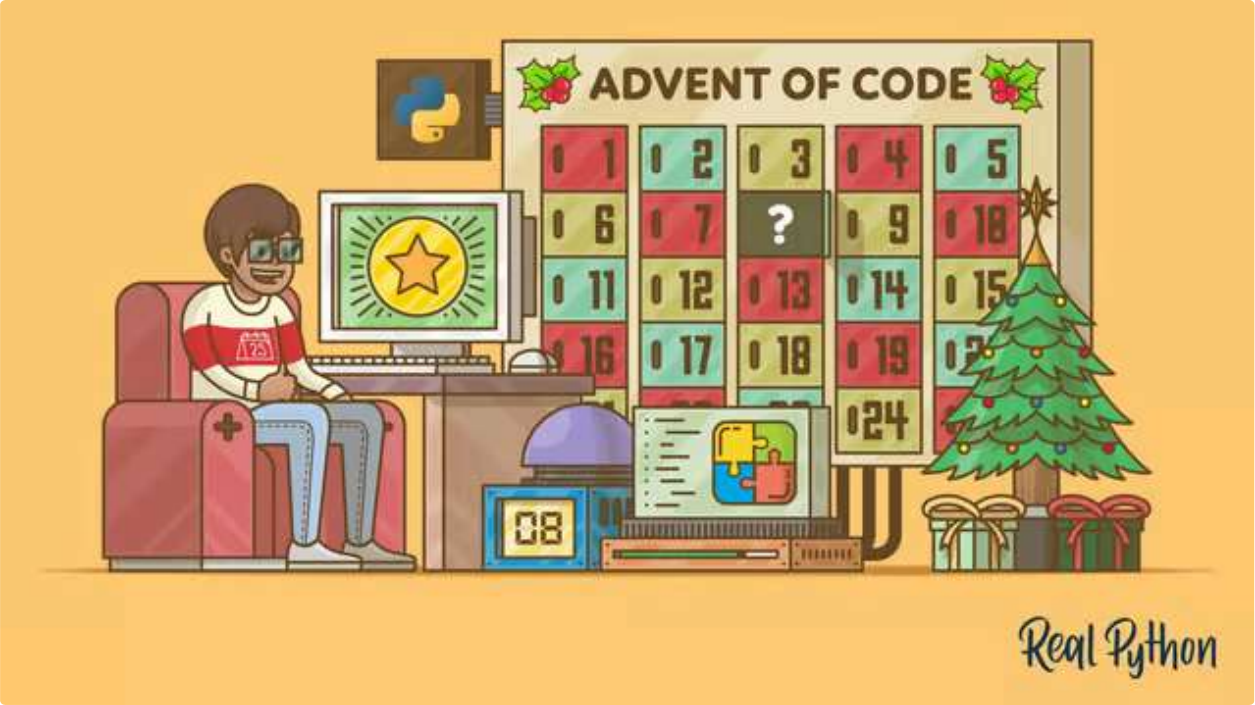

Advent of Code: Solving Your Puzzles With Python

by Geir Arne Hjelle Nov 30, 2022 0 Comments basics career testing

Mark as Completed

Share Share Email

Table of Contents

Help

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Advent of Code: Solving Puzzles With Python**

Advent of Code is an online Advent calendar where you'll find new programming puzzles offered each day from December 1 to 25. While you can solve the puzzles at any time, the excitement when new puzzles unlock is really something special. You can participate in Advent of Code in any programming language—including Python!

With the help of this tutorial, you'll be ready to start solving puzzles and earning your first gold stars.

In this tutorial, you'll learn:

- What an **online Advent calendar** is
- How solving puzzles can **advance your programming skills**
- How you can **participate** in Advent of Code
- How you can **organize your code and tests** when solving Advent of Code puzzles
- How **test-driven development** can be used when solving puzzles

Advent of Code puzzles are designed to be approachable by anyone with an interest in problem-solving. You don't need a heavy computer science background to participate. Instead, Advent of Code is a great arena for learning new skills and testing out new features of Python.

Source Code: Click here to download the free source code that shows you how to solve Advent of Code puzzles with Python.

Puzzling in Programming?

Working on puzzles may seem like a waste of your available programming time. After all, it seems like you're not really producing anything useful and you're not advancing your current projects forward.

However, there are several advantages to taking some time off to practice with programming puzzles:

- Programming puzzles are usually better specified and more contained than your regular job tasks. They offer you the chance to **practice logical thinking** on problems that are less complex than the ones you typically need to handle in your day job.

- You can often challenge yourself with several similar puzzles. This allows you to **build procedural memory**, much like muscle memory, and get experience with structuring certain kinds of code.

- Puzzles are often designed with an eye towards a solution. They allow you to **learn about and apply algorithms** that are tried and tested and are an important part of any programmer's toolbox.

- For some puzzle solutions, even the greatest supercomputers can be too slow if the algorithm is inefficient. You can **analyze the performance** of your solution and get experience to help you understand when a straightforward method is fast enough and when a more optimized procedure is necessary.

- Most programming languages are well-suited for solving programming puzzles. This gives you a great opportunity to **compare different programming languages** for different tasks. Puzzles are also a great way of getting to know a new programming language or trying out some of the newest features of your favorite language.

On top of all of this, challenging yourself with a programming puzzle is often pretty fun! When you add it all up, setting aside some time for puzzles can be very rewarding.

Exploring Options for Solving Programming Puzzles Online

Luckily, there are many websites where you can find programming puzzles and try to solve them. There are often differences in the kinds of problems these websites present, how you submit your solutions, and what kind of feedback and community the sites can offer. You should therefore take some time to look around and find those that appeal the most to you.

In this tutorial, you'll learn about Advent of Code, including what kind of puzzles you can find there and which tools and tricks you can employ to solve them. However, there are also other places where you can get started solving programming puzzles:

- Exercism has learning tracks in many different programming languages. Each learning track offers coding challenges, small tutorials about different programming concepts, and mentors who give you feedback on your solutions.
- Project Euler has been around for a long time. The site offers hundreds of puzzles, usually formulated as math problems. You can solve the problems in any programming language, and once you've solved a puzzle, you get access to a community thread where you can discuss your solution with others.
- Code Wars offers tons of coding challenges, which they call katas. You can solve puzzles in many different programming languages with their built-in editor and automated tests. Afterward, you can compare your solutions to others' and discuss strategies in the forums.
- HackerRank has great features if you're looking for a job. They offer certifications in many different skills, including problem-solving and Python programming, as well as a job board that lets you show off your puzzle-solving skills as part of your job applications.

There are many other sites available where you can practice your puzzle-solving skills. In the rest of this tutorial, you'll focus on what Advent of Code has to offer.

Preparing for Advent of Code: 25 Fresh Puzzles for Christmas

It's time for Advent of Code! It was started by Eric Wastl in 2015. Since then, a new **Advent calendar** of twenty-five new programming puzzles has been published every December. The puzzles have gotten more and more popular over the years. More than 235,000 people have solved at least one of the puzzles from 2021.

> **Note:** Traditionally, an Advent calendar is a calendar used to count the days of Advent while waiting for Christmas. Over the years, Advent calendars have become more commercial and have lost some of their Christian connection.
>
> Most Advent calendars start on December 1 and end on December 24, Christmas Eve, or December 25, Christmas Day. Nowadays, there are all kinds of Advent calendars available, including LEGO calendars, tea calendars, and cosmetics calendars.

In traditional Advent calendars, you open one door every day to reveal what's inside. Advent of Code mimics this by giving you access to one new puzzle each day from December 1 to December 25. For each puzzle you solve, you'll earn gold stars that are yours to keep.

In this section, you'll get more familiar with Advent of Code and see a glimpse of your first puzzle. Later, you'll look at the details of how you can solve these puzzles and practice solving a few of the puzzles yourself.

Advent of Code Puzzles

Advent of Code is an online Advent calendar where a new puzzle is published every day from December 1 to December 25. Each puzzle becomes available at midnight, US Eastern Time. An Advent of Code puzzle has a few typical characteristics:

- Each puzzle consists of two parts, but the second part isn't revealed until you finish the first part.
- You'll earn one gold star (⭐) for each part that you finish. This means you can earn two stars per day and fifty stars if you solve all the puzzles for one year.
- The puzzle is the same for everyone, but you need to solve it based on personalized input that you get from the Advent of Code site. This means that your answer to a puzzle will be different from someone else's, even if you use the same code to calculate it.

You can participate in a global race to be the first to solve each puzzle. However, this is usually pretty crowded with highly skilled, competitive programmers. Advent of Code is probably going to be more fun if you use it as practice for yourself or if you challenge your friends and coworkers to a small, friendly competition.

To get a feeling for how an Advent of Code puzzle works, consider the Day 1 puzzle of 2020:

> Before you leave, the Elves in accounting just need you to fix your **expense report** (your puzzle input); apparently, something isn't quite adding up.
>
> Specifically, they need you to **find the two entries that sum to `2020`** and then multiply those two numbers together.

Each year, there's a wonderfully silly backstory that binds the puzzles together. The 2020 story describes your attempts at leaving for a well-deserved vacation, now that you've saved Christmas several years in a row. The story usually has no effect on the puzzles, but it's still fun to follow along.

In between the plot elements of the story, you'll find the puzzles themselves. In this example, you're looking for two entries in your puzzle input that sum to 2,020. After the explanation that describes the problem, you'll usually find an example showing the calculations that you're expected to do:

> For example, suppose your expense report contained the following:
>
> ```
> 1721
> 979
> 366
> 299
> 675
> 1456
> ```
>
> In this list, the two entries that sum to `2020` are `1721` and `299`. Multiplying them together produces `1721 * 299 = 514579`, so the correct answer is **`514579`**.

The example shows you the answer for this particular list of numbers. If you were about to jump in and start solving this puzzle, you would now start thinking about how you can find the two entries in any valid list of numbers. Before getting deeper into this puzzle, however, you'll explore how to use the Advent of Code site.

How to Participate in Advent of Code

You've seen an example of an Advent of Code puzzle. Next, you'll learn how you can submit your answer for it. You never submit any code to solve the puzzles. You just submit the answer, which is usually a number or a text string.

In general, you'll follow a series of steps to solve a puzzle on the site:

1. **Log in** on the Advent of Code website. You do this by using your credentials from another service like GitHub, Google, Twitter, or Reddit.
2. **Read** the puzzle text and pay special attention to the given example. You should make sure you understand the solution for the example data.
3. **Download** your personalized input for the puzzle. You'll need this input to find your unique answer to the problem.
4. **Code up your solution**. This is the fun part, which you'll get a lot of practice for in the rest of this tutorial.
5. **Enter your answer** to the puzzle on the puzzle page. If your answer is correct, then you earn a gold star, and part two of the puzzle opens up.
6. **Repeat** steps 2 and 4 for part two of the puzzle. This second part is similar to the first, but it usually adds a twist requiring you to adapt your code.
7. **Enter your second answer** on the puzzle page to earn your second star and finish the puzzle.

Remember, you don't submit any code—only your puzzle answers. This means that Advent of Code puzzles can be solved in any programming language. Many people use Advent of Code to practice and learn a new programming language. Eric Wastl, the creator of Advent of Code, gave a talk in 2019 where he talked about the diverse background and motivation of the people participating, among other things.

> **Note:** There's a leaderboard for Advent of Code. In general, you should **ignore this leaderboard**! It only shows who submitted the first 100 solutions after a puzzle opened up. To have a shot at joining the leaderboard, you need a lot of preparation, dedication, and experience with competitive programming.
>
> Instead, you should look at the **private leaderboards**. These become available after you've logged in, and they give you a chance to invite your friends and colleagues to a more relaxed community. You can choose to score your private leaderboard either based on **when** puzzles were solved or simply based on the **number** of puzzles people have solved.
>
> You can also link your name in the private leaderboards to your GitHub account, which allows you to share your solutions with your friends. You set this up by clicking *Settings* in the menu on the Advent of Code site after you've logged in.

Advent of Code is completely free to use, but there are still a few different ways you can support the project:

- **Share information** about Advent of Code on your social media to get the word out.
- **Help others** by taking part in the r/adventofcode subreddit or other forums.
- **Invite your friends** to take part in Advent of Code, sharing your results on a private leaderboard.
- Donate to Advent of Code. If you do, then you'll get an **AoC++** badge next to your name on the site.

In the next sections, you'll see some suggestions on how you can prepare for solving Advent of Code with Python. There's also an awesome list you can check out that links to many different resources related to Advent of Code, including several other people's solutions.

Solving Advent of Code With Python

The Advent of Code has become an annual highlight for many coders around the world. In 2021, more than 235,000 people submitted their solutions. Since Advent of Code started in 2015, programmers have collected more than ten million stars. Many participants use Python to solve the puzzles.

Well, now it's your turn! Head over to the Advent of Code website and have a look at the latest puzzles. Then, come back to this tutorial to get some tips and help to start solving Advent of Code puzzles with Python.

The Anatomy of a Puzzle

In this section, you'll explore the typical anatomy of an Advent of Code puzzle. Additionally, you'll learn about some tools you can use to interact with it.

Each Advent of Code puzzle is split into two parts. When you start working on a puzzle, you only see the first part. The second part unlocks once you've submitted the correct answer to the first part. This is often a twist on the problem that you solved in the first part. Sometimes, you'll find it necessary to refactor your solution from part one, while other times you can solve the second part quickly based on the work you've already done.

Both parts always use the same puzzle input. You can download your puzzle input from the puzzle page for that day. You'll find a link after the puzzle description.

> **Note:** As mentioned earlier, your puzzle input is personalized. This means that if you discuss solutions with other people, their final answers will likely be different from yours.

Everything you need to do in order to submit your puzzle solutions—except actually solving the puzzle—you can do from the Advent of Code website. You should use it to submit your first solutions so that you can get familiar with the flow.

Later, there are several tools that you can use to organize your Advent of Code setup and work more efficiently. For example, you can use the `advent-of-code-data` package to download data. It's a Python package that you can install with `pip`:

Shell

```
$ python -m pip install advent-of-code-data
```

You can use `advent-of-code-data` to download a particular puzzle input set on the command line with its `aocd` tool. Another fun possibility is automatically downloading and caching your personalized puzzle input within your Python code:

Python

```
>>> from aocd.models import Puzzle
>>> puzzle = Puzzle(year=2020, day=1)

>>> # Personal input data. Your data will be different.
>>> puzzle.input_data[:20]
'1753\n1858\n1860\n1978\n'
```

You need to set your session ID in either an environment variable or a file before you can download your personalized data with `advent-of-code-data`. You'll find an explanation for this in the documentation. If you're interested, then you can also use `advent-of-code-data` or `aocd` to submit your solutions and review your earlier answers.

As part of the puzzle text, you'll also find one or several examples typically calculated based on smaller data than your personalized input data. You should read these examples carefully and make sure you understand what you're asked to do before you start coding.

You can use the examples to set up tests for your code. One way is to manually run your solution on the example data and confirm that you're getting the expected answer. Alternatively, you can use a tool like `pytest` to automate the process.

> **Note:** Test-driven development (TDD) is a process where you write tests before implementing your code. Because Advent of Code provides you with expected answers to small examples, it gives you a great opportunity to try out test-driven development on your own.
>
> You'll learn more about TDD later when you try to solve some puzzles by yourself.

You can solve all Advent of Code puzzles using just plain Python and the standard library. However, there are a few packages that can aid you as you're putting together your solutions:

- `advent-of-code-data` can download your input data and submit your solutions.
- `advent-of-code-ocr` can convert the ASCII art solutions of some puzzles to strings.
- `pytest` can check your solution on the example data automatically.
- `parse` can parse strings with a simpler syntax than regular expressions.
- `numpy` can effectively compute with arrays of numbers.
- `colorama` can animate your solutions in the terminal.
- `rich` can render your terminal output more visually appealing.

If you create a virtual environment and install those packages, then you'll have a very solid toolbox for your Advent of Code adventures. Later, you'll see examples of how you can use `parse`, `numpy`, and `colorama` to solve puzzles.

The Structure of a Solution

In the last section, you got familiar with how to read and understand Advent of Code puzzles. In this section, you'll learn how you can solve them. You don't need to do a lot of setup before you solve the Advent of Code puzzles.

Have you thought about how you'd solve the puzzle that you saw earlier? Recall that you're looking for the product of the two numbers in a list that sum to 2,020. Before moving on, think about—and maybe code up—how you'd find which two entries of the following list sum to 2,020:

Python

```
>>> numbers = [1721, 979, 366, 299, 675, 1456]
```

The following script shows one way to solve this first part of the Day 1, 2020, puzzle:

Python

```
1 >>> for num1 in numbers:
2 ...     for num2 in numbers:
3 ...         if num1 < num2 and num1 + num2 == 2020:
4 ...             print(num1 * num2)
5 ...
6 514579
```

The nested `for` loop finds all combinations of two numbers from the list. The test on line 3 is actually slightly more complicated than it needs to be: you only need to test that the numbers sum to 2,020. However, by adding the condition that `num1` should be smaller than `num2`, you avoid solutions being found twice.

In this example, one solution looks like `num1 = 1721` and `num2 = 299`, but since you can add numbers in any order, that means that `num1 = 299` and `num2 = 1721` also form a solution. With the extra check, only the latter combination is reported.

Once you have this solution in place, you can copy your personalized input data into the `numbers` list and calculate your answer to the puzzle.

Note: There are more efficient ways of calculating this answer than trying all possibilities. However, it's usually a good idea to start with a basic approach. Joe Armstrong is quoted as saying:

> Make it work, then make it beautiful, then if you really, really have to, make it fast. 90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful! (Source)
>
> — *Joe Armstrong*

Now that you've seen a working solution for this puzzle, can you make it beautiful?

As you're working through more puzzles, you might start feeling that copying your data into your code and rewriting it into valid Python gets tiresome. Similarly, adding a few functions to your code gives you more flexibility later. You could use them to add tests to your code, for example.

Python has many powerful features for parsing strings. In the long run, you'll be better off leaving the input data just as you downloaded them and letting Python parse them into a usable data structure. In fact, dividing your code into two functions is often beneficial. One function will parse the string input, and the other will solve the puzzle. Based on these principles, you can rewrite your code:

Python

```python
# aoc202001.py

import pathlib
import sys

def parse(puzzle_input):
    """Parse input."""
    return [int(line) for line in puzzle_input.split()]

def part1(numbers):
    """Solve part 1."""
    for num1 in numbers:
        for num2 in numbers:
            if num1 < num2 and num1 + num2 == 2020:
                return num1 * num2

if __name__ == "__main__":
    for path in sys.argv[1:]:
        print(f"\n{path}:")
        puzzle_input = pathlib.Path(path).read_text().strip()

        numbers = parse(puzzle_input)
        print(part1(numbers))
```

On lines 12 to 15, you'll recognize your solution from earlier. First of all, you've wrapped it in a function. This makes it easier to add automatic tests to your code later. You've also added a `parse()` function that can convert lines of strings into a list of numbers.

On line 20, you use `pathlib` to read the contents of a file as text and strip off any blank lines at the end. Looping through `sys.argv` gives you all the filenames entered at the command line.

These changes give you more flexibility as you're working on your solution. Say that you've stored the example data in a file called `example.txt` and your personalized input data in a file named `input.txt`. You can then run your solution on any one of them, or even both, by supplying their names on the command line:

Shell

```
$ python aoc202001.py example.txt input.txt
example.txt:
514579

input.txt:
744475
```

`514579` is indeed the answer to the problem when using the example input data. Remember, the solution for your personalized input data will be different from the one shown above.

Now it's time to give the Advent of Code website a spin! Go to the 2020 Advent of Code calendar and find the puzzle for Day 1. If you haven't already, download your input data and calculate your solution to the puzzle. Then, enter your solution on the website and click *Submit*.

Congratulations! You've just earned your first star!

A Starting Template

As you've seen above, Advent of Code puzzles follow a set structure. Therefore, it makes sense to create a template for yourself that you can use as a starting point when you start to code up a solution. Exactly how much structure you want in such a template is a matter of personal taste. To get started, you'll explore one example of a template that's based on the principles you saw in the previous section:

Python

```python
# aoc_template.py

import pathlib
import sys

def parse(puzzle_input):
    """Parse input."""

def part1(data):
    """Solve part 1."""

def part2(data):
    """Solve part 2."""

def solve(puzzle_input):
    """Solve the puzzle for the given input."""
    data = parse(puzzle_input)
    solution1 = part1(data)
    solution2 = part2(data)

    return solution1, solution2

if __name__ == "__main__":
    for path in sys.argv[1:]:
        print(f"{path}:")
        puzzle_input = pathlib.Path(path).read_text().strip()
        solutions = solve(puzzle_input)
        print("\n".join(str(solution) for solution in solutions))
```

The template has separate functions for parsing the input as well as for solving both parts of a puzzle. You don't need to touch lines 15 to 28 at all. They take care of reading text from an input file, calling `parse()`, `part1()`, and `part2()`, and then reporting the solutions to the console.

You can create a similar template for testing your solutions.

> **Note:** As you learned earlier, the example data are useful for creating tests, as they represents known data with corresponding solutions.

The following template uses `pytest` as a test runner. It's prepared for several different tests, testing each of the functions `parse()`, `part1()`, and `part2()`:

Python

```python
# test_aoc_template.py

import pathlib
import pytest
import aoc_template as aoc

PUZZLE_DIR = pathlib.Path(__file__).parent

@pytest.fixture
def example1():
    puzzle_input = (PUZZLE_DIR / "example1.txt").read_text().strip()
    return aoc.parse(puzzle_input)

@pytest.fixture
def example2():
    puzzle_input = (PUZZLE_DIR / "example2.txt").read_text().strip()
    return aoc.parse(puzzle_input)

@pytest.mark.skip(reason="Not implemented")
def test_parse_example1(example1):
    """Test that input is parsed properly."""
    assert example1 == ...

@pytest.mark.skip(reason="Not implemented")
def test_part1_example1(example1):
    """Test part 1 on example input."""
    assert aoc.part1(example1) == ...

@pytest.mark.skip(reason="Not implemented")
def test_part2_example1(example1):
    """Test part 2 on example input."""
    assert aoc.part2(example1) == ...

@pytest.mark.skip(reason="Not implemented")
def test_part2_example2(example2):
    """Test part 2 on example input."""
    assert aoc.part2(example2) == ...
```

You'll see an example of how you can use this template later. Until then, there are a few things you should note:

- As indicated on line 1, you should name your `pytest` files with a `test_` prefix.
- Similarly, each test is implemented in a function named with a `test_` prefix. You can see examples of these on lines 20, 25, 30, and 35.
- You should change the import on line 5 to import your solution code.
- The template assumes that the example data are stored in files named `example1.txt` and `example2.txt`.
- You should remove the skip marks on lines 19, 24, 29, and 34 when you're ready to start testing.
- You'll need to fill in the ellipses (`...`) on lines 22, 27, 32, and 37 according to the example data and the corresponding solutions.

For example, if you were to adapt this template to the rewritten solution of the first part of the Day 1, 2020, puzzle from the previous section, then you'd need to create a file, `example1.txt`, with the following contents:

Text

```
1721
979
366
299
675
1456
```

Next, you'd remove the skip marks for the first two tests and implement them as follows:

Python

```
# test_aoc202001.py

def test_parse_example1(example1):
    """Test that input is parsed properly."""
    assert example1 == [1721, 979, 366, 299, 675, 1456]

def test_part1_example1(example1):
    """Test part 1 on example input."""
    assert aoc.part1(example1) == 514579
```

Finally, you'd need to make sure that you're importing your solution. If you used the filename `aoc202001.py`, then you should change line 5 to import `aoc202001`:

Python

```
1 # test_aoc202001.py
2
3 import pathlib
4 import pytest
5 import aoc202001 as aoc
6
7 # ...
```

You would then run `pytest` to check your solution. If you implemented your solution correctly, then you'd see something like this:

Shell

```
$ pytest
====================== test session starts ======================
collected 4 items

test_aoc202001.py ..ss                                    [100%]
================= 2 passed, 2 skipped in 0.02s =================
```

Note the two dots (`..`) in front of `ss`. They represent two tests that passed. If the tests had failed, you'd see `F` instead of each dot, along with a detailed explanation of what went wrong.

Tools like Cookiecutter and Copier make it easier to work with templates like these. If you install Copier, then you can use a template similar to the one you've seen here by running the following command:

Shell

```
$ copier gh:gahjelle/template-aoc-python advent_of_code
```

This will set up the template for one particular puzzle in a subdirectory of the `advent_of_code` directory on your computer.

Solution Strategies

Advent of Code puzzles are very diverse. As you advance through the calendar, you'll solve many different problems and discover many different strategies for approaching them.

Some of these strategies are quite general and can be applied to any puzzle. If you find that you're stuck on a puzzle, here are some things you can try to get unstuck:

- **Reread the description**. Advent of Code puzzles are typically very well specified, but some of them can be quite information heavy. Make sure you're not missing a vital part of the puzzle.

- Use the **example data** actively. Make sure you understand how those results are achieved, and check that your code is able to reproduce those examples.
- Some puzzles may get a bit involved. **Break the problem into smaller steps**, and implement and test each step individually.
- If your code works for the example data but not for your personalized input data, then you can build **additional test cases** based on numbers that you're able to calculate by hand to see whether your code covers all corner cases.
- If you're still stuck, then reach out to **your friends** and other puzzle solvers on some of the **forums** dedicated to Advent of Code and ask for hints about how they've solved the puzzle.

As you do more and more puzzles, you'll start to recognize some general kinds of puzzles that come up again and again.

Some puzzles deal with **text and passwords**. Python has several powerful tools for manipulating text strings, including many string methods. To read and parse strings, it's helpful to know the basics of regular expressions. However, you can often get very far with the third-party `parse` library as well.

For example, say that you have the string `"shiny gold bags contain 2 dark red bags."` and want to parse the relevant information from it. You can use `parse` and its pattern syntax:

Python

```
>>> import parse
>>> PATTERN = parse.compile(
...     "{outer_color} bags contain {num:d} {inner_color} bags."
... )

>>> match = PATTERN.search("shiny gold bags contain 2 dark red bags.")
>>> match.named
{'outer_color': 'shiny gold', 'num': 2, 'inner_color': 'dark red'}
```

In the background, `parse` builds a regular expression, but you use a simpler syntax similar to the one that f-strings use.

In some of these text problems, you're explicitly asked to work with **code and parsers**, often building a small custom assembly language. After parsing the code, you often need to run the given program. In practice, this means that you build a small state machine that can track its current state, including the contents of its memory.

You can use classes to keep state and behavior together. In Python, data classes are great for quickly setting up a state machine. In the following example, you implement a small state machine that can handle two different instructions:

Python

```python
# aoc_state_machine.py

from dataclasses import dataclass

@dataclass
class StateMachine:
    memory: dict[str, int]
    program: list[str]

    def run(self):
        """Run the program."""
        current_line = 0
        while current_line < len(self.program):
            instruction = self.program[current_line]

            # Set a register to a value
            if instruction.startswith("set "):
                register, value = instruction[4], int(instruction[6:])
                self.memory[register] = value

            # Increase the value in a register by 1
            elif instruction.startswith("inc "):
                register = instruction[4]
                self.memory[register] += 1

            # Move the line pointer
            current_line += 1
```

The two instructions `set` and `inc` are parsed and handled within `.run()`. Note that the type hints on lines 7 and 8 use a newer syntax that only works on Python 3.9 and later versions. If you're using an older version of Python, then you can import `Dict` and `List` from `typing` instead.

To run your state machine, you first initialize it with an initial memory and load the program into the machine. Next, you call `.run()`. When the program is done, you can inspect `.memory` to see the new state of your machine:

Python

```
>>> from aoc_state_machine import StateMachine
>>> state_machine = StateMachine(
...     memory={"g": 0}, program=["set g 45", "inc g"]
... )
>>> state_machine.run()
>>> state_machine.memory
{'g': 46}
```

This program first set `g` to the value of `45`, then increased it, leaving it at its final value of `46`.

Some fun puzzles involve **grids and labyrinths**. If your grid has a fixed size, then you can use NumPy to get an effective representation of it. Labyrinths are often useful to visualize. You can use Colorama to draw directly in your terminal:

Python

```python
# aoc_grid.py

import numpy as np
from colorama import Cursor

grid = np.array(
    [
        [1, 1, 1, 1, 1],
        [1, 0, 0, 0, 1],
        [1, 1, 1, 0, 1],
        [1, 0, 0, 2, 1],
        [1, 1, 1, 1, 1],
    ]
)

num_rows, num_cols = grid.shape
for row in range(num_rows):
    for col in range(num_cols):
        symbol = " *o"[grid[row, col]]
        print(f"{Cursor.POS(col + 1, row + 2)}{symbol}")
```

This script shows an example of storing a grid using a NumPy array and then using `Cursor.POS` from Colorama to position the cursor in the terminal to print out the grid. When you run this script, you'll see an output like the following:

Shell

```
$ python aoc_grid.py
*****
*   *
*** *
*  o*
*****
```

Visualizing your code as it runs can be fun and also give you some good insights. It can also be an invaluable help when you're debugging and don't quite understand what's happening.

So far in the tutorial, you've gotten some general tips on how you can work with Advent of Code puzzles. In the next sections, you'll get more explicit and solve three puzzles from earlier years.

Practicing Advent of Code: Day 1, 2019

The first puzzle you'll attempt to solve on your own is Day 1, 2019, called **The Tyranny of the Rocket Equation**. This is a typical Day 1 puzzle in that the solution isn't very involved. It's a great exercise to get used to how Advent of Code works and to check that your environment is properly set up.

Part 1: Puzzle Description

In the 2019 story line, you're rescuing Santa, who's become stranded at the edge of the solar system. In the first puzzle, you're getting your rocket ready for launch:

> The Elves quickly load you into a spacecraft and prepare to launch.
>
> At the first Go / No Go poll, every Elf is Go until the Fuel Counter-Upper. They haven't determined the amount of fuel required yet.
>
> Fuel required to launch a given **module** is based on its **mass**. Specifically, to find the fuel required for a module, take its mass, divide by three, round down, and subtract 2.

The example data look like this:

> - For a mass of `12`, divide by 3 and round down to get `4`, then subtract 2 to get `2`.
> - For a mass of `14`, dividing by 3 and rounding down still yields `4`, so the fuel required is also `2`.
> - For a mass of `1969`, the fuel required is `654`.
> - For a mass of `100756`, the fuel required is `33583`.

You need to calculate the total fuel requirements for your spacecraft:

> The Fuel Counter-Upper needs to know the total fuel requirement. To find it, individually calculate the fuel needed for the mass of each module (your puzzle input), then add together all the fuel values.
>
> **What is the sum of the fuel requirements** for all of the modules on your spacecraft?

Now it's time to try to solve the puzzle on your own! It's probably the most fun to download your personalized input data and check your solution on Advent of Code so that you can earn your stars. However, feel free to solve the puzzle based on the example data provided above if you're not ready to sign in to Advent of Code yet.

Part 1: Solution

Once you're done with the puzzle and you've earned your star, you can expand the collapsed block to see a discussion of the puzzle solution:

Solution for Day 1, 2019, Part 1 Show/Hide

You've now solved the first part of the puzzle. However, before moving on to the second part of the puzzle, the next section shows how you can use the templates you saw earlier when solving this problem.

Part 1: Solution Using Templates

Expand the collapsed block below to see another solution to the first part of the Advent of Code puzzle for Day 1, 2019—this time using the templates you saw earlier to organize your code and simplify testing:

Templated solution for Day 1, 2019, Part 1 Show/Hide

You can now move on to the second part of the puzzle. Are you ready for the twist?

Part 2: Puzzle Description

Every Advent of Code puzzle consists of two parts, where the second part is revealed only after you solve the first part. The second part is always related to the first and will use the same input data. However, you may often need to rethink your approach to the first half of the puzzle in order to account for the second half.

Expand the collapsed block below to have a look at the second part of the Advent of Code puzzle for Day 1, 2019:

Day 1, 2019, Part 2 Show/Hide

You'll see a possible solution to part two in the next section. However, try to solve the puzzle for yourself first. If you need a hint to get started, then expand the box below:

Hint for Day 1, 2019, Part 2 Show/Hide

How did you do? Is your rocket ready for launch?

Part 2: Solution

This section shows how you can solve part two, continuing with the template you saw above:

Solution for Day 1, 2019, Part 2 Show/Hide

Congratulations! You've now solved an entire Advent of Code puzzle. Are you ready for a more challenging one?

Practicing Advent of Code: Day 5, 2020

The second puzzle that you'll attempt to solve is the one for Day 5, 2020, called **Binary Boarding**. This puzzle is a bit more challenging than the previous one, but the final solution won't require a lot of code. Start by having a look at the puzzle description for part one.

Part 1: Puzzle Description

In 2020, you're trying hard to get to your well-deserved vacation spot. On Day 5, you're about to board your plane when trouble ensues:

You board your plane only to discover a new problem: you dropped your boarding pass! You aren't sure which seat is yours, and all of the flight attendants are busy with the flood of people that suddenly made it through passport control.

You write a quick program to use your phone's camera to scan all of the nearby boarding passes (your puzzle input); perhaps you can find your seat through process of elimination.

Instead of zones or groups, this airline uses **binary space partitioning** to seat people. A seat might be specified like `FBFBBFFRLR`, where `F` means "front", `B` means "back", `L` means "left", and `R` means "right".

The first 7 characters will either be `F` or `B`; these specify exactly one of the **128 rows** on the plane (numbered `0` through `127`). Each letter tells you which half of a region the given seat is in.

Start with the whole list of rows; the first letter indicates whether the seat is in the **front** (`0` through `63`) or the **back** (`64` through `127`). The next letter indicates which half of that region the seat is in, and so on until you're left with exactly one row.

For example, consider just the first seven characters of `FBFBBFFRLR`:

- Start by considering the whole range, rows `0` through `127`.
- `F` means to take the **lower half**, keeping rows `0` through `63`.
- `B` means to take the **upper half**, keeping rows `32` through `63`.
- `F` means to take the **lower half**, keeping rows `32` through `47`.
- `B` means to take the **upper half**, keeping rows `40` through `47`.
- `B` keeps rows `44` through `47`.
- `F` keeps rows `44` through `45`.
- The final `F` keeps the lower of the two, **row `44`**.

The last three characters will be either `L` or `R`; these specify exactly one of the **8 columns** of seats on the plane (numbered `0` through `7`). The same process as above proceeds again, this time with only three steps. `L` means to keep the **lower half**, while `R` means to keep the **upper half**.

For example, consider just the last 3 characters of `FBFBBFFRLR`:

- Start by considering the whole range, columns `0` through `7`.
- `R` means to take the **upper half**, keeping columns `4` through `7`.
- `L` means to take the **lower half**, keeping columns `4` through `5`.
- The final `R` keeps the upper of the two, **column `5`**.

So, decoding `FBFBBFFRLR` reveals that it is the seat at **row `44`, column `5`**.

Every seat also has a unique **seat ID**: multiply the row by 8, then add the column. In this example, the seat has ID `44 * 8 + 5 =` **`357`**.

Here are some other boarding passes:

- `BFFFBBFRRR`: row `70`, column `7`, seat ID `567`.
- `FFFBBBFRRR`: row `14`, column `7`, seat ID `119`.
- `BBFFBBFRLL`: row `102`, column `4`, seat ID `820`.

As a sanity check, look through your list of boarding passes. **What is the highest seat ID on a boarding pass?**

There's a lot of information in this puzzle description! However, most of it concerns how binary space partitioning works for this particular airline.

Now, try to solve the puzzle for yourself! Keep in mind that if you consider it from the right perspective, the conversion from a boarding pass specification to a seat ID isn't as complicated as it might seem at first. If you find that you're struggling with that part, then expand the box below to see a hint on how you can get started:

Hint for Day 5, 2020, Part 1 Show/Hide

When you're done with your solution, have a look in the next section to see a discussion about the puzzle.

Part 1: Solution

Now that you've given it a shot yourself, you can go ahead and expand the following block to see one way that you could solve the puzzle:

Solution for Day 5, 2020, Part 1 Show/Hide

Time to move on to the second part of the puzzle. Will you be able to board the plane?

Part 2: Puzzle Description

Expand the section below when you're ready for the second part of the puzzle:

Day 5, 2020, Part 2 Show/Hide

Take your time and work on your solution to this second part.

Part 2: Solution

Open the box below when you're ready to compare your solution to another one:

Solution for Day 5, 2020, Part 2 Show/Hide

Great, you've finished another puzzle! To round things out, try your hand at one of the puzzles from 2021.

Practicing Advent of Code: Day 5, 2021

As a third example of an Advent of Code puzzle, you'll look closely at Day 5, 2021. The puzzle is called **Hydrothermal Venture** and will take you on a deep-sea adventure. The solution will be a bit more involved than the previous two puzzles. Check out the puzzle description.

Part 1: Puzzle Description

The story line in 2021 starts with the Elves carelessly dropping the keys to Santa's sleigh into the ocean. You end up in a submarine searching for them in order to save Christmas. On day 5, you come across a field of hydrothermal vents on the ocean floor.

It turns out that these vents are harmful to your submarine and you need to map out the field to avoid the most dangerous areas:

They tend to form in **lines**; the submarine helpfully produces a list of nearby lines of vents (your puzzle input) for you to review. For example:

Text

```
0,9 -> 5,9
8,0 -> 0,8
9,4 -> 3,4
2,2 -> 2,1
7,0 -> 7,4
6,4 -> 2,0
0,9 -> 2,9
3,4 -> 1,4
0,0 -> 8,8
5,5 -> 8,2
```

Each line of vents is given as a line segment in the format `x1,y1 -> x2,y2` where `x1,y1` are the coordinates of one end the line segment and `x2,y2` are the coordinates of the other end. These line segments include the points at both ends. In other words:

- An entry like `1,1 -> 1,3` covers points `1,1`, `1,2`, and `1,3`.
- An entry like `9,7 -> 7,7` covers points `9,7`, `8,7`, and `7,7`.

For now, **only consider horizontal and vertical lines**: lines where either `x1 = x2` or `y1 = y2`.

The example shows how the puzzle input describes lines at given coordinates. Your job is to find where these lines overlap:

To avoid the most dangerous areas, you need to determine **the number of points where at least two lines overlap**. In the above example, this is [...] a total of **5** points.

Consider only horizontal and vertical lines. **At how many points do at least two lines overlap?**

As in the previous puzzle, there's a lot of information in the puzzle text. The information mostly pertains to how you should interpret your puzzle input.

Note: There's some additional information in the full puzzle description. In particular, there's a diagram that shows all the lines drawn on a grid.

Try to solve the puzzle for yourself. When you're done, move on to the next sections to see one possible solution.

Part 1: Input Parsing

There are many ways to implement a solution to this puzzle. Expand the block below to start working with the input data:

Parse input for Day 5, 2021, Part 1 Show/Hide

Once you've represented the data in a structure that you can work with, then you can move on to solving the puzzle itself.

Part 1: Solution

You'll continue working on part 1 of the puzzle. The following solution takes advantage of the structural pattern matching feature introduced in Python 3.10. Expand the collapsed section to read the details:

Solution for Day 5, 2021, Part 1 Show/Hide

Phew! You've finished the first part of the puzzle. Time to look at what part 2 has in store for you.

Part 2: Puzzle Description

Expand the section below to read the second part of the puzzle for Day 5, 2021:

Day 5, 2021, Part 2 Show/Hide

Play around with the second part and try to solve it yourself. Once you’re done, have a look in the next section for a possible solution.

Part 2: Solution

Click to reveal the solution below when you’re ready to have a look at a solution and compare it with your own:

Solution for Day 5, 2021, Part 2 Show/Hide

Congratulations! By now, you’ve solved at least three Advent of Code puzzles. Luckily, there are hundreds more waiting for you!

Conclusion

Advent of Code is a great resource of fun programming puzzles! You can use it to practice your problem-solving skills and challenge your friends to a fun competition and common learning experience. You can hear even more about Advent of Code in the following episode of the Real Python Podcast: Solving Advent of Code Puzzles With Python.

If you haven’t already done so, then head over to the Advent of Code website and try out some of the new puzzles.

In this tutorial, you’ve learned:

- How solving puzzles can **advance your programming skills**
- How you can **participate** in Advent of Code
- How you can approach **different kinds** of puzzles
- How you can **organize your code and tests** when solving Advent of Code puzzles
- How **test-driven development** can be used when solving puzzles

Real Python hosts a private leaderboard and a community forum about Advent of Code. Become a Real Python member and join the `#advent-of-code` Slack channel to access it.

Source Code: Click here to download the free source code that shows you how to solve Advent of Code puzzles with Python.

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Advent of Code: Solving Puzzles With Python**