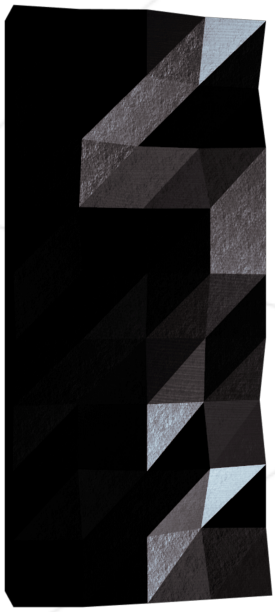

Pacman

Ghosts! More ghosts!

Summary: Recreate the famous arcade game Pac-man!

Version: 1.0

Contents

Chapter I

Forewords

First released in 1980 by Namco, **Pac-Man** quickly became a cultural icon and one of the most influential video games of all time. Designed by Toru Iwatani, its goal was to create a game that could appeal to women and casual players, contrasting with the space shooters of the era. The game introduced the now-famous ghost AI, each with unique behaviors (Blinky chases, Pinky ambushes, Inky is unpredictable, and Clyde is... weird).

Pac-Man was also the first game to popularize the concept of a power-up — the Pacgum (or Power Pellets) that lets you eat the ghosts. The original arcade machine had 256 levels, but due to an integer overflow bug, level 256 was impossible to finish, known as the infamous "kill screen".

With this project, you'll breathe new life into this classic by building your own version — in Python, with modern structure and project organization, ready to be deployed on a real gaming platform. Waka-waka!

Chapter II

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Common Instructions

III.1 General Rules

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- Your functions should handle exceptions gracefully to avoid crashes. Use `try-except` blocks to manage potential errors. Prefer context managers for resources like files or connections to ensure automatic cleanup. If your program crashes due to unhandled exceptions during the review, it will be considered non-functional.
- All resources (e.g., file handles, network connections) must be properly managed to prevent leaks. Use context managers where possible for automatic handling.
- Your code must include type hints for function parameters, return types, and variables where applicable (using the `typing` module). Use `mypy` for static type checking. All functions must pass mypy without errors.
- Include docstrings in functions and classes following PEP 257 (e.g., Google or NumPy style) to document purpose, parameters, and returns.

III.2 Makefile

Include a `Makefile` in your project to automate common tasks. It must contain the following rules (mandatory lint implies the specified flags; it is strongly recommended to try `-strict` for enhanced checking):

- **install**: Install project dependencies using `pip`, `uv`, `pipx`, or any other package manager of your choice.
- **run**: Execute the main script of your project (e.g., via Python interpreter).
- **debug**: Run the main script in debug mode using Python's built-in debugger (e.g., pdb).
- **clean**: Remove temporary files or caches (e.g., __pycache__, .mypy_cache) to keep the project environment clean.

- **lint**: Execute the commands `flake8 .` and `mypy . --warn-return-any --warn-unused-ignores --ignore-missing-imports --disallow-untyped-defs --check-untyped-defs`
- **lint-strict** (optional): Execute the commands `flake8 .` and `mypy . --strict`

III.3 Additional Guidelines

- Create test programs to verify project functionality (not submitted or graded). Use frameworks like `pytest` or `unittest` for unit tests, covering edge cases.
- Include a `.gitignore` file to exclude Python artifacts.
- It is recommended to use virtual environments (e.g., venv or conda) for dependency isolation during development.

If any additional project-specific requirements apply, they will be stated immediately below this section.

Chapter IV

Mandatory part

In this project, you will create a complete and playable **Pac-Man** game like in Python, using object-oriented programming, a simple graphical library (MLX or similar), and a modular, reusable architecture.

The game must support:

- A custom configuration via a file (JSON with comments) to set game parameters.
- Robust error handling, no crash!
- Level generation based on an external ‘A-Maze-ing‘ package (not yours!).
- A persistent highscore system of your choice (stored in a json file on project, saved to disk, etc.).
- A polished graphical UI with main menu, game view, and game-over handling.
- A cheat mode for evaluation purposes.
- Deployment to a public gaming platform (Steam/Itch.io or similar) for demonstration.

Special care will be given to project management, and multiple documents will be requested.

Game Loop

- Main Menu > start game > Win or Lose > Enter name for highscore > Back to Main Menu

Chapter V

Detailed specifications

V.1 Usage

Your program must be launched from the command line as follows:

```
$> python3 pac-man.py config.json
```

- It must take **exactly one** argument: a configuration file. The filename file does not matter, but the file must be a json file.
- Any error (missing file, invalid value, missing key) must be handled cleanly with clear messages, **never** with a Python traceback.

V.2 Configuration file

The config file uses JSON. In addition to the standard JSON format, you must handle comments. Lines starting with `#` are comments and must be ignored. You may also support additional comment styles (e.g., C or C++).
The exact structure is up to you, but document your keys in the README and provide robust defaults.

Suggested keys (names are indicative):

- `highscore_filename`
- `level` array of multiple levels
- `width` , `height` for each level
- `lives` : 3
- `pacgum` : 42
- `points_per_pacgum` : 10
- `points_per_super_pacgum` : 50
- `points_per_ghost` : 200

- `seed : 42`
- `level_max_time : 90`

V.3 Faulty config handling

On missing or invalid values, clamp to safe defaults, log a clear message, and continue. Unknown keys must be ignored. No traceback.

`The game configuration will be updated during the defense.`

V.4 Maze generator integration

You **must not** write your own generator.
At project start, you are assigned an *A-Maze-ing* package from another group:

- You must use their package **as-is**, without modifying it. The assigned package will be re-installed during your peer review.
- Your loader must adapt to their interface, not the opposite.
- The parameter `PERFECT` will be set to `False` produce Pac-Man-compatible corridors (see A-Maze-ing subject).
- If the generator fails, you must handle the error cleanly.

V.5 Highscore system

You must implement a persistent highscore system.
The exact implementation is up to you, but document it in the README.

It must:

- Be stored as you choose (e.g., json file on project, on disk, etc.).
- Be robust to file errors (missing file, invalid format, etc.).
- Handle player names (max 10 characters, alphanumeric and spaces only).
- Handle scores (non-negative integers).

- Keep the top 10 highscores, with player names and scores.
- Load highscores at game start and save them at game end.
- Allow players to enter their name when they end the game (win or lose).
- Display highscores in the main menu.

Chapter VI

Game specifications

VI.1 Level structure

- A maze with walls and corridors, generated by the assigned A-Maze-ing package.
- The first level is composed of a maze generated with a fixed seed (e.g., 42), then, each subsequent level is composed of a maze randomly generated.
- Pacgums (small dots) in most corridors.
- Super-pacgums (power pellets) in the 4 corners of the maze.
- 4 ghosts, one in each corner of the maze.
- The player starts in the middle of the maze.

VI.2 Player

- Can move through corridors only (no walls).
- Can move in 4 directions (up, down, left, right) using arrow keys or WASD (depending on your keyboard).
- Starts with 3 lives.
- Loses a life when touched by a ghost.
- Respawns in the middle of the maze after losing a life.
- Game over when all lives are lost.
- Wins the level when all pacgums are eaten.
- Wins the game when all levels are completed.
- Eating a pacgum increases the score by X points.
- Eating a super-pacgum (power pellet) increases the score by Y points and makes ghosts edible for a short time.

- Eating an edible ghost increases the score by Z points.

VI.3 Ghosts

- Move autonomously through corridors.
- Chase the player when not edible. It's up to you to define the chase behavior (distance-based, random, etc.).
- Run away from the player when edible.
- Respawn to their corner after a while when eaten (e.g., after 5 or 10 seconds).

VI.4 Pacgums and Super-pacgums

- Pacgums are small dots placed in most corridors.
- Super-pacgums (power pellets) are larger dots placed in the 4 corners of the maze.
- Eating a pacgum increases the score by X points.
- Eating a super-pacgum increases the score by Y points and makes ghosts edible for a short time.

VI.5 Cheat mode

- Can be activated to facilitate peer review.
- Suggested features:
  - Invincibility (no life lost; ghosts cannot eat the player).
  - Level skip (immediately win the current level).
  - Ghost freeze (ghosts stop moving).
  - Extra lives (add extra lives to the player).
  - Increased speed (player moves faster).
  - Any other feature that may be useful.

Keep in mind that the cheat mode is for peer review purposes, and therefore it must genuinely help the reviewer to test all features of your game easily.

VI.6 Scoring

- The score increases when:
  - Eating a pacgum (+X points).
  - Eating a super-pacgum (+Y points).
  - Eating an edible ghost (+Z points).
- The score does not decrease.

VI.7 Game progression

- The game consists of multiple levels (at least 10).
- Each level has a time limit (e.g., 90 seconds).
- If the time limit is reached, you can decide what happens (e.g., restart the level, end the game, etc.).
- If the player completes a level, they move to the next level.
- The player keeps their score and remaining lives between levels.
- The game ends when all levels are completed or when the player loses all lives.
- During the game, the player can pause and resume the game.
- When the game ends (win or lose), the final score is displayed, and the player can enter their name to save the highscore.
- After the game ends, the player is returned to the main menu.

VI.8 User Interface

- Main Menu:
  - Start Game - Allows the player to start a new game.
  - View Highscores - Displays the **Top 10** scores with player names.
  - Instructions - Shows the game controls and rules.
  - Exit - Closes the game application.
- In-Game HUD (always visible during gameplay):
  - Current score
  - Remaining lives
  - Current level
  - Remaining time for the level
- Pause Menu:
  - Resume the game - Allows the player to return to the ongoing game.
  - Return to the main menu - Allows the player to exit the current game and go back to the main menu.
- Game Over Screen:
  - Displays the final score.
  - Prompts the player to enter their name to save the score in the highscores list.
- Victory Screen:
  - Displays the final score and a congratulatory message.
  - Prompts the player to enter their name to save the score in the highscores list.

Chapter VII

Project packaging

You must deliver:

- A complete game that can be installed and launched from a public platform (e.g., Steam or Itch.io) as a free but unlisted/private build.
- The packaged game must be fully functional.
- Provide minimal in-package instructions(controls, options, configuration).
- Your Git repository must contain the full source and the packaging script/spec at the root.
- You may be asked to regenerate the package during the peer review.

Chapter VIII

Project management

During this project, you must use a structured approach to drive the project and provide evidence. You are free to use any approach and tools you may need to support your project management. You must produce various documents that prove your approach. In a dedicated subdirectory of your repository, include any relevant elements.

Examples:

- Project timeline, Gantt, Kanban, etc.
- Actual progress tracking, compared to the timeline.
- Project analysis, and associated choices.
- Risk analysis, and possible mitigation.
- Team organization (who did what, how decisions were made, how issues were handled).
- Acceptance test plan (features tested, bugs found/fixed).
- Summary of any blocking points or conflicts during the project.
- etc.

Chapter IX

Readme Requirements

A `README.md` file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.
The `README.md` must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A "**Description**" section that clearly presents the project, including its goal and a brief overview.
- An "**Instructions**" section containing any relevant information about compilation, installation, and/or execution.
- A "**Resources**" section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.

➠ **Additional sections may be required depending on the project** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.

- A **Configuration** section explaining the config file structure and default values.
- A **Highscore** section explaining how the highscore system works and why you decided to implement it this way.
- A **Maze Generation** section explaining how the assigned A-Maze-ing package is used to generate mazes.
- an **Implementation** section with a technical summary of your implementation.
- A **General Software Architecture** section, with high-level overview of the software architecture (modules, classes, and their relationships).

- A **Project Management** section, with a brief overview of how you managed the project and a link to the dedicated project management directory.

Chapter X

Submission and peer-review

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be reviewed during the defense. Don't hesitate to double-check the names of your files to ensure they are correct.

Recode instructions

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behaviour change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific time frame is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.