

PROGRAMMING with AI (V2)

Updated May 2026

Transparency is required.

Where generative AI is used to produce code for assessed work, you must be specific about what was produced by the AI, what changes you made to it, and what you wrote yourself. You should do this using inline comments in the code, using the appropriate notation for the programming language you are using. Just like using code from other external sources (textbooks, GitHub repositories, stack overflow etc.), there should be no doubt in the marker's mind which parts of the work were completed by you and which parts were produced by generative AI.

Lecturers may give more detailed guidance.

In some cases, lecturers may permit you to use generative AI to solve more complex problems. They will communicate with you if this is the case. This document should serve as a default position, which lecturers may supersede. If you are ever unclear about what is and is not acceptable for a given assignment, ask your module lecturer for specific help.

You may find that early in your programming education, you may not be able to make good use of generative AI because the problems you are solving are correspondingly simple and may only be trivialised using AI. As you progress, you will solve larger problems and it will become more appropriate to use generative AI to produce boilerplate code or to solve smaller sub-problems.

RAG Grids for AI Tool Usage

Some lecturers may use a red/amber/green system to give detailed guidance on how AI tools may be used for different parts of an assessment. For example, an assessment that requires the development of some code to analyse a synthetic dataset and a report documenting the code might be accompanied by the following grids:

FOR THE CODE			
Use	Permitted?	Advice	How to acknowledge use
To write code	No	It is important that you design and build your code yourself. You need to understand how a language and its data structures work in practice before you can clearly identify if an AI tool is doing it wrong.	
To debug code	With caution	AI tools can help identify syntax errors or suggest possible fixes, but do not	Add the comment /*This section of code was debugged using <TOOL>

		rely on them. Always read and understand the explanation, test the correction yourself, and make sure that you learn why the bug occurred.	on <DATE>*/ wherever you have used an AI tool to help debug your code
To develop test cases	Yes	AI tools can help suggest possible test cases, including edge cases you may not have considered. Review and refine these tests yourself, making sure that they match the program specification and genuinely verify correctness.	Add the comment /*This test case was developed using <TOOL> on <DATE>*/ on every test case you've developed with the aid of an AI tool.
To comment code	Yes	Although AI tools are often quite good at commenting code you should always make sure that the comments explain what is happening at that point in the code.	Add the comment /*This code was commented by <TOOL> on <DATE>*/ to each file which has been commented using a AI tool.

If the assessment requires a written report then it is good practice to include a subsection on AI Use to document how AI has been used in the assessment. If the assessment requires a presentation then you should include a similar slide on AI Use. These should identify how AI tool(s) were used in the creation of the code AND how it was used in the creation of the related report/presentation. Lecturers may provide a similar grid outlining the permitted use of AI tools in writing the report and/or presentation.

Recognising the value in learning programming

Generative AI tools are now capable of writing functional code in response to a simple text prompt, which raises a reasonable question: why should you invest time and effort in learning to program yourself? Using AI-generated code responsibly still requires a genuine understanding of programming. Without this foundation, you will find it very difficult to judge whether the code an AI produces is correct, efficient, or even safe to use. AI tools make mistakes – they may generate code that appears plausible but contains subtle logical errors, uses inappropriate methods, or fails entirely on inputs that differ from those you tested with. A programmer who understands the language and the problem can catch these errors, while a student who cannot read code cannot.

There is also a deeper issue of transferable skill. Learning to program is not simply about memorising syntax; it develops problem-solving strategies, logical thinking, and the ability to break complex tasks down into manageable steps. These are skills that AI cannot develop for you, and they will serve you across your entire career. Understanding what a program does and why it works the way it does will make you a far more effective user of AI tools in the long run – you will know how to prompt them well, how to evaluate their output critically, and how to extend or adapt the code they produce for your own purposes.

This is why, at Level 4, you are expected to write code largely yourself, with only limited and carefully declared use of generative AI. This is not an arbitrary restriction: it reflects the same principle that applies in any discipline where foundational skills must be established before more advanced tools can be used responsibly. A medical student must learn anatomy before using diagnostic AI; an engineer must understand structural mechanics before relying on simulation software. In the same way, a student using code in their discipline must first develop the ability to write, read, and reason about programs independently. As you progress through the levels of your programme, you will be permitted to make increasing use of generative AI, reflecting your growing ability to direct, evaluate, and take responsibility for the code it produces.

Generative AI within Integrated Development Environments

This guidance applies equally to query-driven generative AI such as ChatGPT and Bard, as it does to “autocomplete” generative AI within an integrated development environment (IDE), such as GitHub co-pilot, Tabnine or IntelliCode. In either instance, these tools should be used judiciously and with caution. Violating this policy inadvertently using an autocomplete generative AI or similar tool still constitutes academic misconduct.

What you cannot do

You **cannot and must not** simply prompt an AI tool to write code on the topic of your assignment.

If we assume that your task is to write a Python script to analyse survey data collected for a research project.

You **cannot and must not** input a prompt like this to an AI tool:

- Write me a Python script that analyses my survey data and produces a summary report.
- Give me examples of Python scripts that analyse survey data, with an explanation of what they do.
- Read the assignment brief included and provide a solution.

If you do this, then the code is no longer yours. If you submit work that is not your own, then you will be committing **academic misconduct**, which carries severe penalties.

You will find it difficult to use generative AI for very simple problems without trivialising that problem and risking committing academic misconduct. What your lecturer may allow you to do includes:

Ask questions to explore the problem.

You can ask generative AI specific questions which inform the development of your code, or aid in your understanding of the topic. For example:

- What is the difference between a list and a dictionary in Python?
- How does a for loop work in Python?
- What does the pandas library do, and how do I use it to read a CSV file?
- What is a mean, and how do I calculate the mean of a column of numbers in Python?

These sorts of questions are always acceptable because they do not produce any code but explain concepts which will inform your development. These sorts of questions will usually not yield answers beyond the capabilities of a search engine but may allow a more probing contextual discussion.

Snippets

Aligned with common programming practices, it may be appropriate to prompt generative AI for small snippets of code. In the previous example, this might be to generate boilerplate, e.g.:

- Write a Python class with three numeric attributes: wellbeing, engagement, and satisfaction. Create an `__init__` method and a display method.

However, within the scope of the example, a prompt like this would **not be acceptable**:

- Write a Python class to store survey responses, calculate average scores, and classify each response as High, Moderate, or Low.

It is appropriate to use generative AI to create boilerplate code, or to generate solutions to trivial sub-problems. You should avoid using generative AI to trivialise significant components of the work you have been required to undertake.

This may, at times, be a difficult distinction to make, you should always ask your lecturer if unsure.

Debugging

Several AI debugging tools are now available, capable of detecting errors in code and making suggestions for their remedy. These tools are broadly comparable with static code analysis tools and are permitted – however you must again be transparent about their use.

Example

Combining the above, if we wanted to create a vector class as a component of our graphics engine, we could prompt our generative AI engine:

> Write a Python class with three numeric attributes: wellbeing, engagement, and satisfaction. Create an `__init__` method and a display method.

The generative AI tool responds:

Sure! Here is a Python class with the requested attributes and methods:

Python

```
class SurveyResponse:
    def __init__(self, wellbeing, engagement, satisfaction):
        self.wellbeing = wellbeing
        self.engagement = engagement
        self.satisfaction = satisfaction

    def display(self):
```

```
print(f"Wellbeing: {self.wellbeing}, Engagement: {self.engagement}, Satisfaction: {self.satisfaction}")
```

```
# Example usage
```

```
response = SurveyResponse(4, 3, 5)
```

```
response.display()
```

```
response.wellbeing = 5
```

```
response.display()
```

This code creates a SurveyResponse object with scores of 4, 3, and 5, displays them, updates the wellbeing score to 5, and displays again.

We should now indicate that this is not our original work by adding an inline comment. In Python, comments begin with the # character. In this case:

```
# The boilerplate for this class was created through generative AI (ChatGPT 5.4). # Specifically, the class
```

```
# structure, wellbeing, engagement and satisfaction attributes, the __init__
```

```
# constructor and the
```

```
# display method were produced in this way.
```

Having trivialised some of the boilerplate code with generative AI, we can then manually add the more advanced functionality ourselves. In this case, I will add an average_score method and a classify method that categorises a response as High, Moderate, or Low based on the average of the three scores. I could write code such as:

```
def average_score(self):  
    return (self.wellbeing + self.engagement + self.satisfaction) / 3
```

```
def classify(self):  
    avg = self.average_score()  
    if avg >= 4:  
        return "High"  
    elif avg >= 2.5:  
        return "Moderate"  
    else:  
        return "Low"
```

Then update the comment:

```
# The boilerplate for this class was created through generative AI (ChatGPT). Specifically, the class  
# structure, wellbeing, engagement and satisfaction attributes, the __init__ constructor and the  
# display method were produced in this way.
```

```
# The average_score and classify methods were created from scratch without the use of generative AI.
```

Giving a final implementation as:

survey_analysis.py

```
# The boilerplate for this class was created through generative AI (ChatGPT).
# Specifically, the class structure, wellbeing, engagement and satisfaction attributes,
# the __init__ constructor and the display method were produced in this way.

# The average_score and classify methods were created from scratch without the use of generative AI.

class SurveyResponse:
    def __init__(self, wellbeing, engagement, satisfaction):
        self.wellbeing = wellbeing
        self.engagement = engagement
        self.satisfaction = satisfaction

    def display(self):
        print(f"Wellbeing: {self.wellbeing}, Engagement: {self.engagement}, Satisfaction: {self.satisfaction}")

    def average_score(self):
        return (self.wellbeing + self.engagement + self.satisfaction) / 3

    def classify(self):
        avg = self.average_score()
        if avg >= 4:
            return "High"
        elif avg >= 2.5:
            return "Moderate"
        else:
            return "Low"

# Example usage
response = SurveyResponse(4, 3, 5)
response.display()
print(f"Average score: {response.average_score():.1f}")
print(f"Classification: {response.classify()}")
```

This is acceptable because it:

- ✓ Uses generative AI to produce boilerplate code, not solve complex problems.
- ✓ Makes use of the generative AI implementation to add meaningful functionality.
- ✓ Is wholly transparent about what generative AI was used to do and what was produced by the human developer.