Clear and uncluttered mainline
The main() function clearly shows the step-by-step process of creating each component, collecting input, and assembling the bike. It avoids cluttered logic and long nested structures. Keeping the mainline readable and sequential makes the overall logic easy to understand, debug, and verify which is especially helpful for future updates.

```python
def main():
    print("\nWelcome to the MTB Builder")

    frame = Frame(...)
    fork = Fork(...)
    shock = Shock(...)
    ...
```

One logical task per subroutine
Subroutines like input_number() and choose_from_menu() each perform one clear task: either validating numeric input or letting the user choose from a list. This modularity makes each function easier to test and reuse, and avoids bugs caused by mixed responsibilities.

```python
def input_number(prompt, allow_float=False):
    while True:
        try:
            value = input(prompt + ": ")
            return float(value) if allow_float else int(value)
        except ValueError:
            print("Invalid input.")
```

Use of stubs
Stub functions like validate_component() and backup_build() were included as placeholders during development to allow testing without requiring full implementations. Stubs allow you to build and test other program components before those parts are fully implemented or finalised.

```python
def validate_component(component):
    return True

def backup_build(bike):
    print("\n[Stub] Simulated backup: Build not yet saved to file.")
```

Use of control structures and data structures
I used while loops to loop through lists like drivetrain parts or extras, and stored components in data structures (e.g. lists, objects) that represent real-world bike parts. This keeps logic intuitive and avoids errors. Using control structures in a consistent way improves program flow and maintainability.

```python
extras = []
while True:
    name = input("Extra name (or 'done'): ")
    if name.lower() == 'done':
        break
    brand = input(f"{name} brand: ")
    price = input_number(f"'{name}' price", allow_float=True)
    extras.append(Extra(name, brand, price))
```

Ease of maintenance
Class and method names are self-explanatory, such as total_price(), Fork, Shock, and choose_from_menu(). Code is logically structured and comments are included where needed. Well-named variables and methods make the code readable to others, making debugging and enhancements easier later.

```python
class Component:
    def __init__(self, name, brand, price):
        self.name = name
        self.brand = brand
        self.price = float(price)
```

Version control
I used version control by first editing a file called prototype.py and then uploading the changes to my main file Bike_component.py to ensure that I always had a version of the code that works to go back to. While not using Github, manual version control allowed me to try new features without risking the original file, supporting safe experimentation and rollback.

Regular backup

The stub backup_build() was included to simulate saving builds to file. I also made sure to save both of my files to a folder called mu_code in my icloud drive to ensure the code was saved and easily accessible. This protects against data loss due to hardware failure.

```python
def backup_build(bike):
    print("\n[Stub] Simulated backup: Build not yet saved to file.")
```