Palindromic DNA sequences

Get sequence from user

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Get input from user

```python
def get_sequence() -> str | None:
    sequence_prompt = "Enter a DNA sequence or path to single-sequence DNA FASTA

    while not (sequence := extract_seq_from_input(input(sequence_prompt))):
        print("Input type is not acceptable.")

        retry_prompt = "Would you like to retry? ([Y]/N) "

        while (should_retry := input(retry_prompt).upper()) not in "YN":
            continue

        match should_retry:
            case "Y" | "":
                continue
            case "N":
                return None

    return sequence
```

Get input from user

```python
def get_sequence() -> str | None:
    sequence_prompt = "Enter a DNA sequence or path to single-sequence DNA FASTA

    while not (sequence := extract_seq_from_input(input(sequence_prompt))):
        print("Input type is not acceptable.")

        retry_prompt = "Would you like to retry? ([Y]/N) "

        while (should_retry := input(retry_prompt).upper()) not in "YN":
            continue

        match should_retry:
            case "Y" | "":
                continue
            case "N":
                return None

    return sequence
```

Get input from user

```python
def get_sequence() -> str | None:
    sequence_prompt = "Enter a DNA sequence or path to single-sequence DNA FASTA

    while not (sequence := extract_seq_from_input(input(sequence_prompt))):
        print("Input type is not acceptable.")

        retry_prompt = "Would you like to retry? ([Y]/N) "

        while (should_retry := input(retry_prompt).upper()) not in "YN":
            continue

        match should_retry:
            case "Y" | "":
                continue
            case "N":
                return None

    return sequence
```

Get input from user

```python
def get_sequence() -> str | None:
    sequence_prompt = "Enter a DNA sequence or path to single-sequence DNA FASTA

    while not (sequence := extract_seq_from_input(input(sequence_prompt))):
        print("Input type is not acceptable.")

        retry_prompt = "Would you like to retry? ([Y]/N) "

        while (should_retry := input(retry_prompt).upper()) not in "YN":
            continue

        match should_retry:
            case "Y" | "":
                continue
            case "N":
                return None

    return sequence
```

Get input from user

```python
def get_sequence() -> str | None:
    sequence_prompt = "Enter a DNA sequence or path to single-sequence DNA FASTA

    while not (sequence := extract_seq_from_input(input(sequence_prompt))):
        print("Input type is not acceptable.")

        retry_prompt = "Would you like to retry? ([Y]/N) "

        while (should_retry := input(retry_prompt).upper()) not in "YN":
            continue

        match should_retry:
            case "Y" | "":
                continue
            case "N":
                return None

    return sequence
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Check input & extract DNA sequence

```python
def is_dna(sequence: str) -> bool:
    sequence = sequence.upper()
    return bool(sequence) and set(sequence).issubset("ACGT")


def extract_seq_from_input(string: str) -> str | None:
    if string.endswith((".fa", ".fasta", ".fna")):
        fasta_file = string
        with open(fasta_file, "r", encoding="utf8") as f:
            lines = [line.strip() for line in f.readlines()]
            sequence = "".join(
                filter(lambda line: not line.startswith(">"), lines)
            ).upper()
    else:
        sequence = string

    return sequence if is_dna(sequence) else None
```

Continue the program if DNA sequence was extracted successfully

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Continue the program if DNA sequence was extracted successfully

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Ask user for min/max lengths

```python
def get_min_and_max_lengths() -> dict[str, int | float]:

    lengths = {
        "min_palindrome_length": 4,
        "max_palindrome_length": 10,
        "max_spacer_fraction": 0.5,
    }

    for length, default_value in lengths.items():
        length_prompt = f"Please choose {length} [default: {default_value}]: "

        while not is_number(user_input := input(length_prompt)):
            if not user_input:
                break
            else:
                print("Please enter a numeric value.")
                continue

        if user_input:
            lengths[length] = float(user_input)

    return lengths
```

Ask user for min/max lengths

```python
def get_min_and_max_lengths() -> dict[str, int | float]:

    lengths = {
        "min_palindrome_length": 4,
        "max_palindrome_length": 10,
        "max_spacer_fraction": 0.5,
    }

    for length, default_value in lengths.items():
        length_prompt = f"Please choose {length} [default: {default_value}]: "

        while not is_number(user_input := input(length_prompt)):
            if not user_input:
                break
            else:
                print("Please enter a numeric value.")
                continue

        if user_input:
            lengths[length] = float(user_input)

    return lengths
```

Ask user for min/max lengths

```python
def get_min_and_max_lengths() -> dict[str, int | float]:

    lengths = {
        "min_palindrome_length": 4,
        "max_palindrome_length": 10,
        "max_spacer_fraction": 0.5,
    }

    for length, default_value in lengths.items():
        length_prompt = f"Please choose {length} [default: {default_value}]: "

        while not is_number(user_input := input(length_prompt)):
            if not user_input:
                break
            else:
                print("Please enter a numeric value.")
                continue

        if user_input:
            lengths[length] = float(user_input)

    return lengths
```

Ask user for min/max lengths

```python
def get_min_and_max_lengths() -> dict[str, int | float]:

    lengths = {
        "min_palindrome_length": 4,
        "max_palindrome_length": 10,
        "max_spacer_fraction": 0.5,
    }

    for length, default_value in lengths.items():
        length_prompt = f"Please choose {length} [default: {default_value}]: "

        while not is_number(user_input := input(length_prompt)):
            if not user_input:
                break
            else:
                print("Please enter a numeric value.")
                continue

        if user_input:
            lengths[length] = float(user_input)

    return lengths
```

Ask user for min/max lengths

```python
def get_min_and_max_lengths() -> dict[str, int | float]:

    lengths = {
        "min_palindrome_length": 4,
        "max_palindrome_length": 10,
        "max_spacer_fraction": 0.5,
    }

    for length, default_value in lengths.items():
        length_prompt = f"Please choose {length} [default: {default_value}]: "

        while not is_number(user_input := input(length_prompt)):
            if not user_input:
                break
            else:
                print("Please enter a numeric value.")
                continue

        if user_input:
            lengths[length] = float(user_input)

    return lengths
```

Ask user for min/max lengths

```python
def get_min_and_max_lengths() -> dict[str, int | float]:

    lengths = {
        "min_palindrome_length": 4,
        "max_palindrome_length": 10,
        "max_spacer_fraction": 0.5,
    }

    for length, default_value in lengths.items():
        length_prompt = f"Please choose {length} [default: {default_value}]: "

        while not is_number(user_input := input(length_prompt)):
            if not user_input:
                break
            else:
                print("Please enter a numeric value.")
                continue

        if user_input:
            lengths[length] = float(user_input)

    return lengths
```

Ask user for path to output file

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Ask user for path to output file

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Ask user for path to output file

```python
def get_out_file() → str:
    default_out_file = "palindromes.output.txt"

    prompt = f"Enter out_file_path [default: {default_out_file}]: "

    if out_file := input(prompt):
        return out_file

    return default_out_file
```

Ask user for path to output file

```python
def get_out_file() -> str:
    default_out_file = "palindromes.output.txt"

    prompt = f"Enter out_file_path [default: {default_out_file}]: "

    if out_file := input(prompt):
        return out_file

    return default_out_file
```

Ask user for path to output file

```python
def get_out_file() -> str:
    default_out_file = "palindromes.output.txt"

    prompt = f"Enter out_file_path [default: {default_out_file}]: "

    if out_file := input(prompt):
        return out_file

    return default_out_file
```

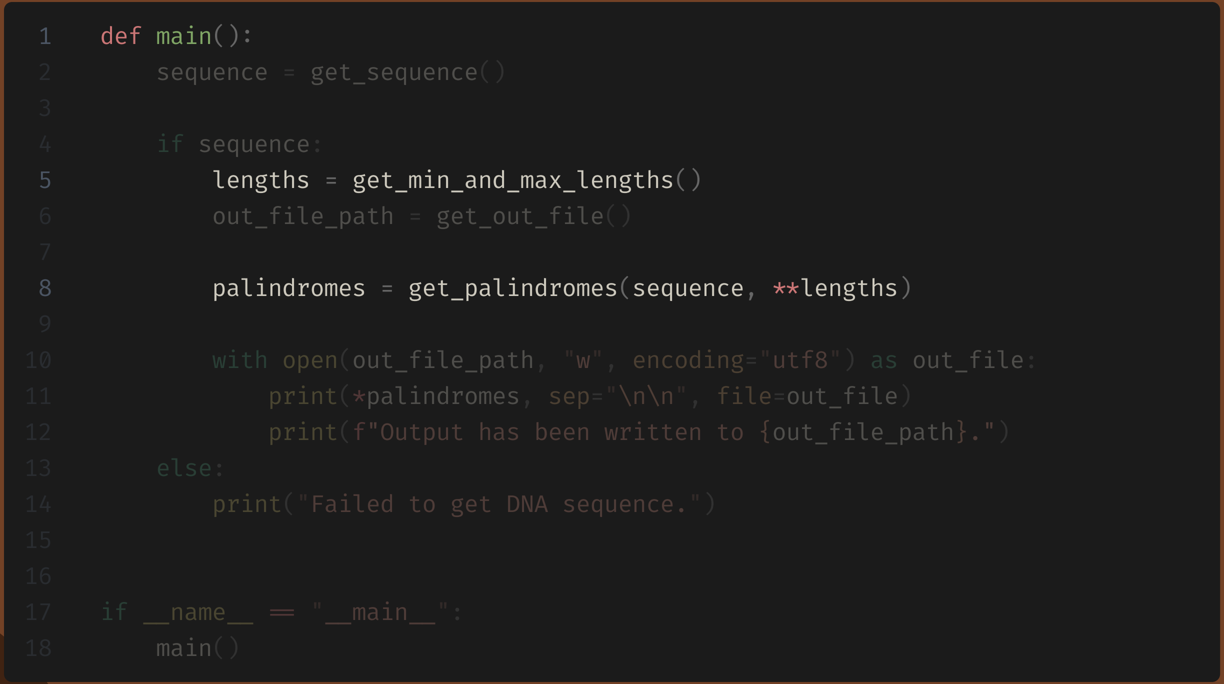
Ask user for path to output file

```python
def get_out_file() -> str:
    default_out_file = "palindromes.output.txt"

    prompt = f"Enter out_file_path [default: {default_out_file}]: "

    if out_file := input(prompt):
        return out_file

    return default_out_file
```

Get palindromes from user sequence

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Get palindromes from user sequence



```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

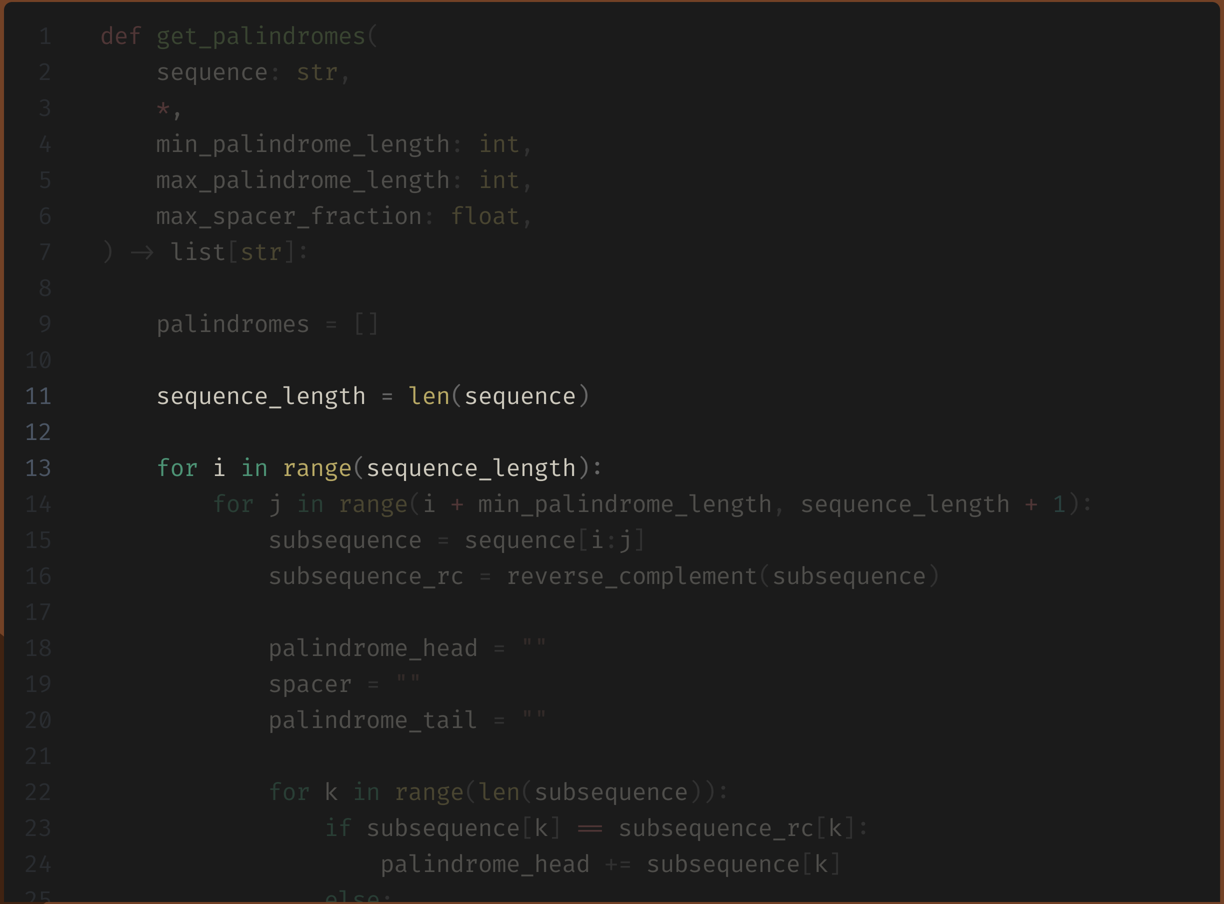
Get palindromes from user sequence

```python
def get_palindromes(
    sequence: str,
    *,
    min_palindrome_length: int,
    max_palindrome_length: int,
    max_spacer_fraction: float,
) -> list[str]:

    palindromes = []

    sequence_length = len(sequence)

    for i in range(sequence_length):
        for j in range(i + min_palindrome_length, sequence_length + 1):
            subsequence = sequence[i:j]
            subsequence_rc = reverse_complement(subsequence)

            palindrome_head = ""
            spacer = ""
            palindrome_tail = ""

            for k in range(len(subsequence)):
                if subsequence[k] == subsequence_rc[k]:
                    palindrome_head += subsequence[k]
                else:
```

Get palindromes from user sequence

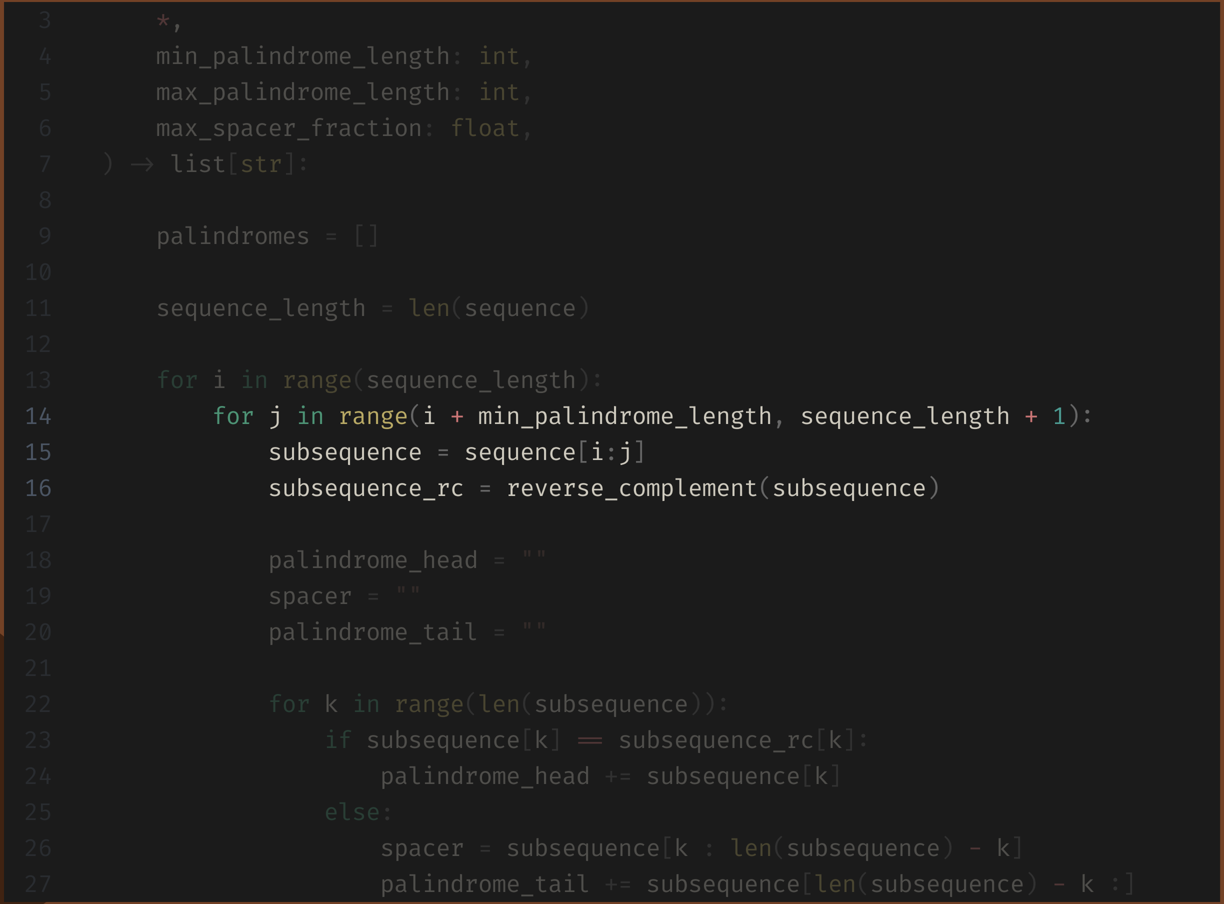
```python
def get_palindromes(
    sequence: str,
    *,
    min_palindrome_length: int,
    max_palindrome_length: int,
    max_spacer_fraction: float,
) -> list[str]:

    palindromes = []

    sequence_length = len(sequence)

    for i in range(sequence_length):
        for j in range(i + min_palindrome_length, sequence_length + 1):
            subsequence = sequence[i:j]
            subsequence_rc = reverse_complement(subsequence)

            palindrome_head = ""
            spacer = ""
            palindrome_tail = ""

            for k in range(len(subsequence)):
                if subsequence[k] == subsequence_rc[k]:
                    palindrome_head += subsequence[k]
                else:
```

Get palindromes from user sequence



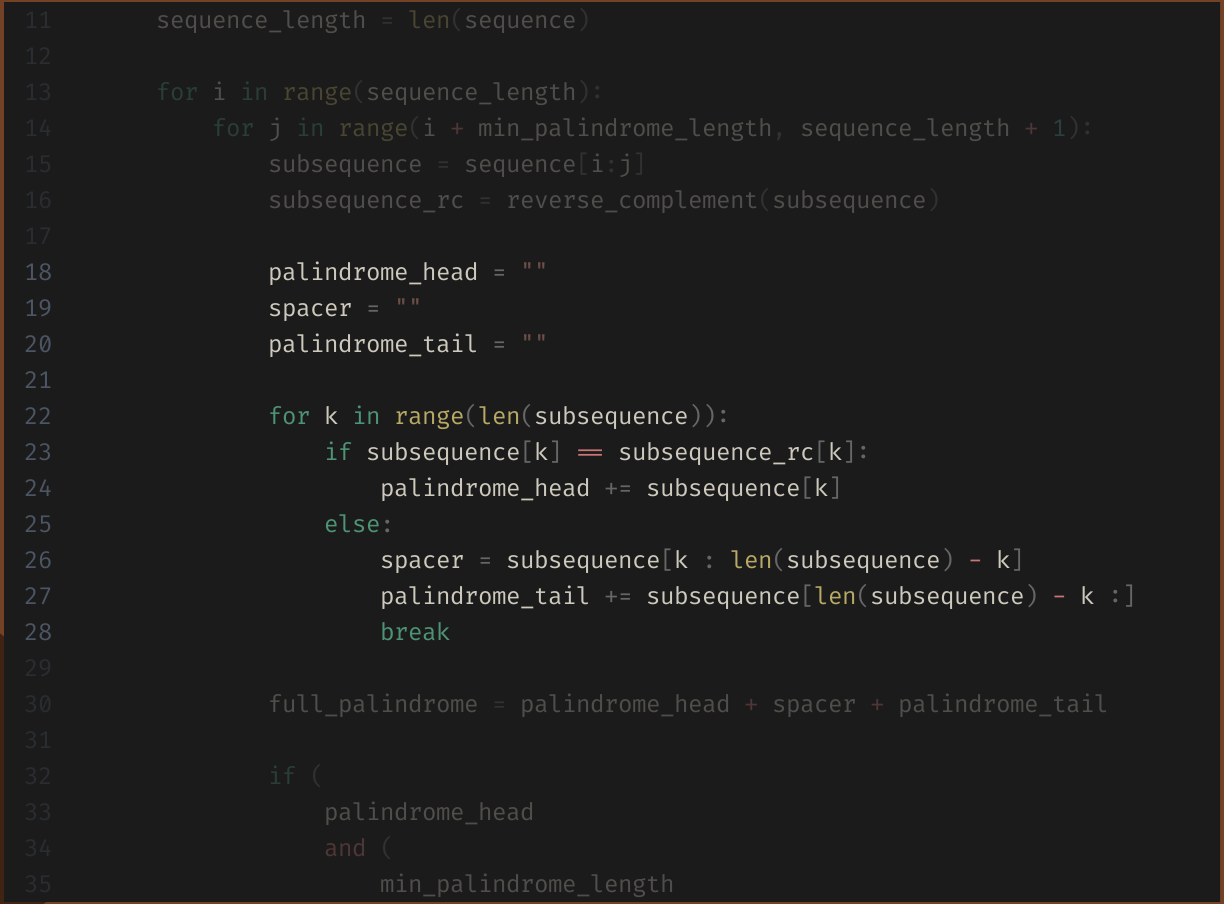
```python
def get_palindromes(
    sequence: str,
    *,
    min_palindrome_length: int,
    max_palindrome_length: int,
    max_spacer_fraction: float,
) -> list[str]:

    palindromes = []

    sequence_length = len(sequence)

    for i in range(sequence_length):
        for j in range(i + min_palindrome_length, sequence_length + 1):
            subsequence = sequence[i:j]
            subsequence_rc = reverse_complement(subsequence)

            palindrome_head = ""
            spacer = ""
            palindrome_tail = ""

            for k in range(len(subsequence)):
                if subsequence[k] == subsequence_rc[k]:
                    palindrome_head += subsequence[k]
                else:
```

Get palindromes from user sequence



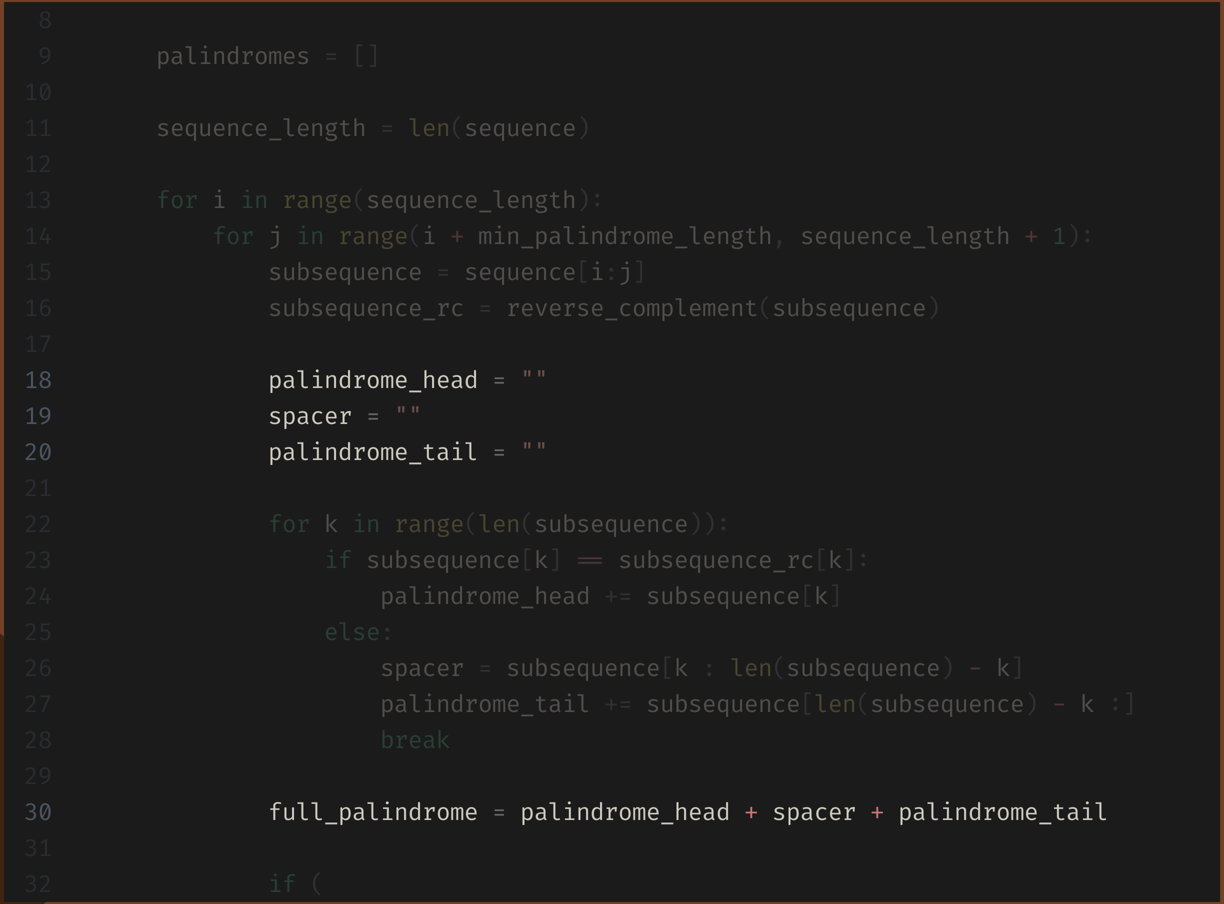
```python
    *,
    min_palindrome_length: int,
    max_palindrome_length: int,
    max_spacer_fraction: float,
) -> list[str]:

    palindromes = []

    sequence_length = len(sequence)

    for i in range(sequence_length):
        for j in range(i + min_palindrome_length, sequence_length + 1):
            subsequence = sequence[i:j]
            subsequence_rc = reverse_complement(subsequence)

            palindrome_head = ""
            spacer = ""
            palindrome_tail = ""

            for k in range(len(subsequence)):
                if subsequence[k] == subsequence_rc[k]:
                    palindrome_head += subsequence[k]
                else:
                    spacer = subsequence[k : len(subsequence) - k]
                    palindrome_tail += subsequence[len(subsequence) - k :]
```

Get palindromes from user sequence



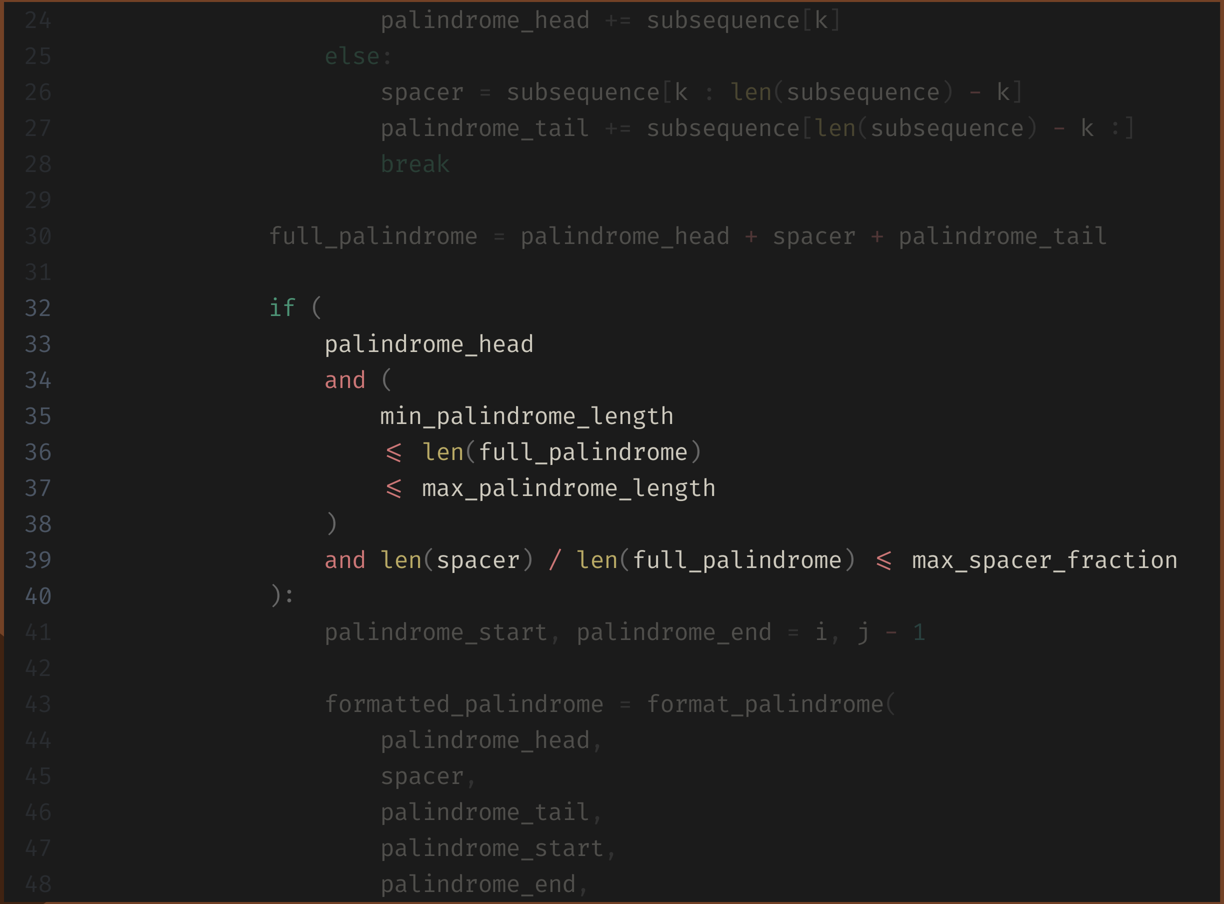
```
    sequence_length = len(sequence)

    for i in range(sequence_length):
        for j in range(i + min_palindrome_length, sequence_length + 1):
            subsequence = sequence[i:j]
            subsequence_rc = reverse_complement(subsequence)

            palindrome_head = ""
            spacer = ""
            palindrome_tail = ""

            for k in range(len(subsequence)):
                if subsequence[k] == subsequence_rc[k]:
                    palindrome_head += subsequence[k]
                else:
                    spacer = subsequence[k : len(subsequence) - k]
                    palindrome_tail += subsequence[len(subsequence) - k :]
                    break

            full_palindrome = palindrome_head + spacer + palindrome_tail

            if (
                palindrome_head
                and (
                    min_palindrome_length
```

Get palindromes from user sequence



```python

    palindromes = []

    sequence_length = len(sequence)

    for i in range(sequence_length):
        for j in range(i + min_palindrome_length, sequence_length + 1):
            subsequence = sequence[i:j]
            subsequence_rc = reverse_complement(subsequence)

            palindrome_head = ""
            spacer = ""
            palindrome_tail = ""

            for k in range(len(subsequence)):
                if subsequence[k] == subsequence_rc[k]:
                    palindrome_head += subsequence[k]
                else:
                    spacer = subsequence[k : len(subsequence) - k]
                    palindrome_tail += subsequence[len(subsequence) - k :]
                    break

            full_palindrome = palindrome_head + spacer + palindrome_tail

            if (
```

Get palindromes from user sequence

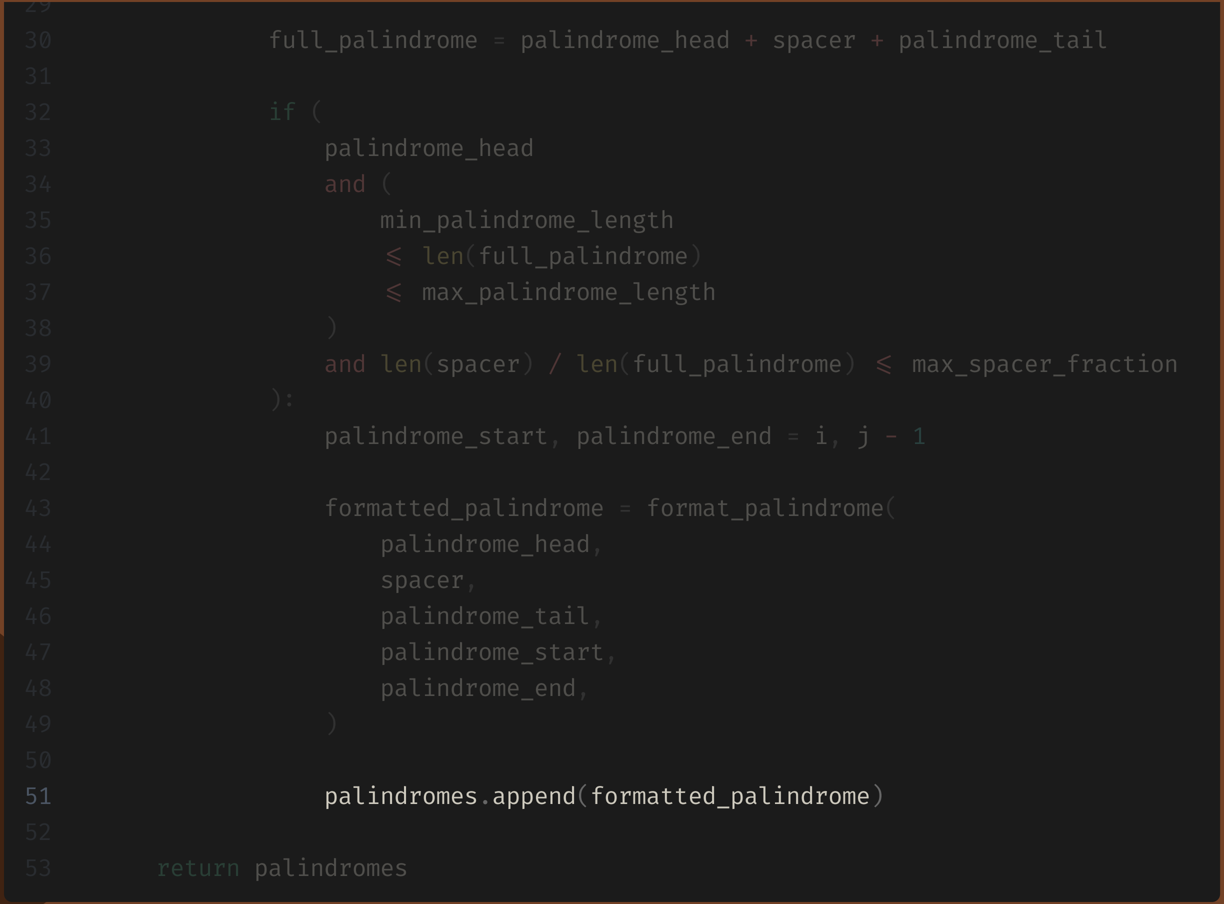


```python
                palindrome_head += subsequence[k]
            else:
                spacer = subsequence[k : len(subsequence) - k]
                palindrome_tail += subsequence[len(subsequence) - k :]
                break

        full_palindrome = palindrome_head + spacer + palindrome_tail

        if (
            palindrome_head
            and (
                min_palindrome_length
                <= len(full_palindrome)
                <= max_palindrome_length
            )
            and len(spacer) / len(full_palindrome) <= max_spacer_fraction
        ):
            palindrome_start, palindrome_end = i, j - 1

            formatted_palindrome = format_palindrome(
                palindrome_head,
                spacer,
                palindrome_tail,
                palindrome_start,
                palindrome_end,
```

Get palindromes from user sequence

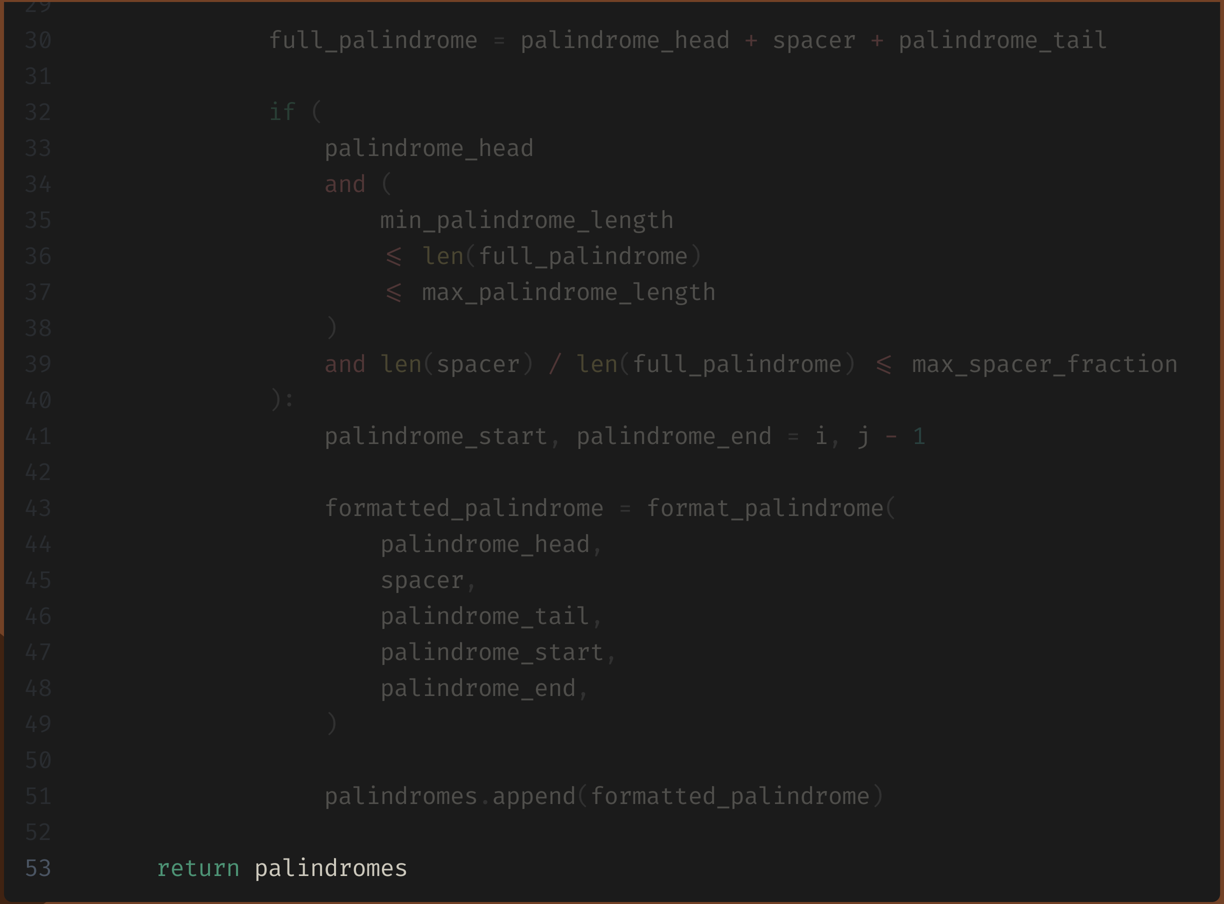
```python
            full_palindrome = palindrome_head + spacer + palindrome_tail

            if (
                palindrome_head
                and (
                    min_palindrome_length
                    ≤ len(full_palindrome)
                    ≤ max_palindrome_length
                )
                and len(spacer) / len(full_palindrome) ≤ max_spacer_fraction
            ):
                palindrome_start, palindrome_end = i, j - 1

                formatted_palindrome = format_palindrome(
                    palindrome_head,
                    spacer,
                    palindrome_tail,
                    palindrome_start,
                    palindrome_end,
                )

                palindromes.append(formatted_palindrome)

    return palindromes
```

Get palindromes from user sequence



```
            full_palindrome = palindrome_head + spacer + palindrome_tail

            if (
                palindrome_head
                and (
                    min_palindrome_length
                    <= len(full_palindrome)
                    <= max_palindrome_length
                )
                and len(spacer) / len(full_palindrome) <= max_spacer_fraction
            ):
                palindrome_start, palindrome_end = i, j - 1

                formatted_palindrome = format_palindrome(
                    palindrome_head,
                    spacer,
                    palindrome_tail,
                    palindrome_start,
                    palindrome_end,
                )

                palindromes.append(formatted_palindrome)

    return palindromes
```

Get palindromes from user sequence



```
            full_palindrome = palindrome_head + spacer + palindrome_tail

            if (
                palindrome_head
                and (
                    min_palindrome_length
                    ⩽  len(full_palindrome)
                    ⩽  max_palindrome_length
                )
                and len(spacer) / len(full_palindrome) ⩽  max_spacer_fraction
            ):
                palindrome_start, palindrome_end = i, j - 1

                formatted_palindrome = format_palindrome(
                    palindrome_head,
                    spacer,
                    palindrome_tail,
                    palindrome_start,
                    palindrome_end,
                )

                palindromes.append(formatted_palindrome)

    return palindromes
```

Print palindromes to output file

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Print palindromes to output file

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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

Format palindromes for easier interpretation

```python
def format_palindrome(
    palindrome_head, spacer, palindrome_tail, palindrome_start, palindrome_end
) -> str:
    full_palindrome = palindrome_head + spacer + palindrome_tail

    bonds = "|" * len(palindrome_head) + "-" * len(spacer) + "|" * len(palindrom

    formatted_palindrome = "\n".join(
        [
            f"{palindrome_start}",
            "\u2193",
            full_palindrome,
            bonds,
            full_palindrome[::-1],
            f"{"\u2191":>{len(full_palindrome)}}",
            f"{palindrome_end:>{len(full_palindrome)}}",
        ],
    )

    return formatted_palindrome
```

Format palindromes for easier interpretation

```python
def format_palindrome(
    palindrome_head, spacer, palindrome_tail, palindrome_start, palindrome_end
) -> str:
    full_palindrome = palindrome_head + spacer + palindrome_tail

    bonds = "|" * len(palindrome_head) + "-" * len(spacer) + "|" * len(palindrom

    formatted_palindrome = "\n".join(
        [
            f"{palindrome_start}",
            "\u2193",
            full_palindrome,
            bonds,
            full_palindrome[::-1],
            f"{"\u2191":>{len(full_palindrome)}}",
            f"{palindrome_end:>{len(full_palindrome)}}",
        ],
    )

    return formatted_palindrome
```

Format palindromes for easier interpretation

```python
def format_palindrome(
    palindrome_head, spacer, palindrome_tail, palindrome_start, palindrome_end
) -> str:
    full_palindrome = palindrome_head + spacer + palindrome_tail

    bonds = "|" * len(palindrome_head) + "-" * len(spacer) + "|" * len(palindrom

    formatted_palindrome = "\n".join(
        [
            f"{palindrome_start}",
            "\u2193",
            full_palindrome,
            bonds,
            full_palindrome[::-1],
            f"{"\u2191":>{len(full_palindrome)}}",
            f"{palindrome_end:>{len(full_palindrome)}}",
        ],
    )

    return formatted_palindrome
```

Format palindromes for easier interpretation

```python
def format_palindrome(
    palindrome_head, spacer, palindrome_tail, palindrome_start, palindrome_end
) -> str:
    full_palindrome = palindrome_head + spacer + palindrome_tail

    bonds = "|" * len(palindrome_head) + "-" * len(spacer) + "|" * len(palindrom

    formatted_palindrome = "\n".join(
        [
            f"{palindrome_start}",
            "\u2193",
            full_palindrome,
            bonds,
            full_palindrome[::-1],
            f"{"\u2191":>{len(full_palindrome)}}",
            f"{palindrome_end:>{len(full_palindrome)}}",
        ],
    )

    return formatted_palindrome
```

Format palindromes for easier interpretation

```python
def format_palindrome(
    palindrome_head, spacer, palindrome_tail, palindrome_start, palindrome_end
) -> str:
    full_palindrome = palindrome_head + spacer + palindrome_tail

    bonds = "|" * len(palindrome_head) + "-" * len(spacer) + "|" * len(palindrom

    formatted_palindrome = "\n".join(
        [
            f"{palindrome_start}",
            "\u2193",
            full_palindrome,
            bonds,
            full_palindrome[::-1],
            f"{"\u2191":>{len(full_palindrome)}}",
            f"{palindrome_end:>{len(full_palindrome)}}",
        ],
    )

    return formatted_palindrome
```

Notify user of success

```python
def main():
    sequence = get_sequence()

    if sequence:
        lengths = get_min_and_max_lengths()
        out_file_path = get_out_file()

        palindromes = get_palindromes(sequence, **lengths)

        with open(out_file_path, "w", encoding="utf8") as out_file:
            print(*palindromes, sep="\n\n", file=out_file)
            print(f"Output has been written to {out_file_path}.")
    else:
        print("Failed to get DNA sequence.")


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