Here are a few more types of custom functions in Python:

11. **Closures**:
    - Functions that "remember" values in the enclosing scope even if they are not present in memory.

    ```python
    def outer_function(x):
        def inner_function(y):
            return x + y
        return inner_function

    closure = outer_function(10)
    result = closure(5)  # Result: 15
    ```

12. **Partial Functions**:
    - Functions that "freeze" some portion of a function's arguments and keywords to create a new function with reduced arity.

    ```python
    from functools import partial

    def power(base, exponent):
        return base ** exponent

    square = partial(power, exponent=2)
    result = square(5)  # Result: 25
    ```

13. **Generator Expressions**:
    - Inline, memory-efficient generators that are similar to list comprehensions but produce values lazily.

    ```python
    squares = (x ** 2 for x in range(1, 6))
    for square in squares:
        print(square)
    ```

14. **Map, Filter, and Reduce Functions**:
    - Built-in functions that operate on iterables, allowing you to apply a function to each element, filter elements based on a condition, or reduce a sequence of values to a single value.

```python
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x ** 2, numbers)
filtered_numbers = filter(lambda x: x % 2 == 0, numbers)
from functools import reduce
product = reduce(lambda x, y: x * y, numbers)
```

15. **Async Functions**:
    - Functions that are defined with the `async` keyword, allowing asynchronous execution with `await` statements.

```python
import asyncio

async def fetch_data(url):
    # Asynchronous code to fetch data from a URL
    pass

async def process_data(data):
    # Asynchronous code to process data
    pass

async def main():
    data = await fetch_data("https://example.com")
    await process_data(data)

asyncio.run(main())
```

16. **Generator Functions with `yield from`**:
    - Advanced generator functions that can delegate part of their operations to another generator.

```python
def generator1():
    yield 1
    yield from generator2()
    yield 3

def generator2():
    yield 2

for value in generator1():
```

```
    print(value)
```