# Advanced Programming

Generator functions

# READING A LARGE FILE

- Assume that we have a very large text file containing billions of lines
- Process this file line by line requires to
  - Load the entire file into memory
    - Inefficient and may not fit into memory
  - Tokenize each line and store them in an ADT (lists or dictionary)
    - Inefficient and may not fit into memory

```
import re

def process_file(input_file, output_file):
    with open(input_file, 'r') as f_input:
        lines = f_input.readlines()
    processed_lines = [tokenize_line(line)
                       for line in lines]
    with open(output_file, 'w') as f_output:
        for line in processed_lines:
            f_output.write(','.join(line) + '\n')

def tokenize_line(line):
    tokens = []
    if len(line) > 2:
        tokens = re.findall(r'\b\w+\b', line)
    return tokens
```

# USING A CALLBACK FUNCTION

- Function passed as an argument to another function
  - Executed at a later time usually through an event
- They are commonly used for asynchronous programming
  - Executes code concurrently and handle events efficiently
  - Triggered on custom events based on conditions

```
def tokenize_file(filename, callback_fn):
  with open(filename) as f:
    for line in f:
      for token in line.split():
        callback_fn(token)

def print_token(token):
  print(f”{token}")

if __name__ == "__main__":
    tokenize_file("t2.txt", print_token)
```

Python, functions are first-class objects - they can be passed as arguments to other functions

# ITERATOR APPROACH

- Make tokenizer as an iterator,
    - Use iterator’s .next() method  to get the next token
    - More user-friendly
    - Burdens tokenizer with the task of maintaining state between invocations

```python
import re

class Tokenizer:
    def __init__(self, input_file):
        self.input_file = input_file
        self.file_handle = open(input_file, 'r')
        self.current_line = None
        self.current_position = 0

    def tokenize_next(self):
        if self.current_line is None or \
        self.current_position >= len(self.current_line):
            self.current_line = self.file_handle.readline()
            if not self.current_line:
                raise StopIteration
            self.current_position = 0

        tokens = re.findall(r'\b\w+\b',
                            self.current_line[self.current_position:])
        if not tokens:
            # No more tokens in the current line,
            # move to the next line
            self.current_line = None
            return self.tokenize_next()

        # Update the position for the next call
        token_length = len(tokens[0])
        self.current_position += token_length
        # Account for spaces between tokens
        while self.current_position < len(self.current_line) and \
            not self.current_line[self.current_position].isalnum():
            self.current_position += 1

        return tokens[0]
```

```python
if __name__ == '__main__':
    input_file = 't2.txt'
    tokenizer = Tokenizer(input_file)
    try:
        while True:
            token = tokenizer.tokenize_next()
            print(token)
    except StopIteration:
        pass
    tokenizer.file_handle.close()
```

# THREAD-BASED TOKENISATION

- Running the producer and consumer in separate threads allows both to maintain their states naturally

```python
import re
import multiprocessing

def tokenize_chunk(chunk, output_queue):
    tokens = []
    for line in chunk:
        tokens.extend(tokenize_line(line))
    output_queue.put(tokens)

def main(input_file, output_file,
         num_processes=4, chunk_size=1000):
    output_queue = multiprocessing.Queue()

    with open(input_file, 'r') as f_input:
        chunks = [f_input.readlines(chunk_size)
                  for _ in range(num_processes)]
    processes = []
    for chunk in chunks:
        process = multiprocessing.Process(target=tokenize_chunk,
                                          args=(chunk, output_queue))
        process.start()
        processes.append(process)

    tokens = []
    for _ in range(num_processes):
        tokens.extend(output_queue.get())

    for process in processes:
        process.join()

    with open(output_file, 'w') as f_output:
        for token in tokens:
            f_output.write(token + '\n')
```

```python
def tokenize_line(line):
    return re.findall(r'\b\w+\b', line)
```

```python
if __name__ == "__main__":
    input_file = 't2.txt'
    output_file = 'output.txt'
    main(input_file, output_file)
```

# IS THERE ANY ALTERNATE SOLUTION?

- Develop a functionality that
  - Returns an intermediate result
  - Gets the value to its caller (consumer)
  - Maintains the local state
  - Resumes right where it stopped the computation

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b


if __name__ == '__main__':
    fib = fibonacci()
    for _ in range(10):
        print(next(fib))
```

# WHAT IS A GENERATOR FUNCTION?

- A function that iterates over a potentially infinite sequence of values without creating the entire sequence in memory at once

# GENERATOR FUNCTION

- Uses the `yield` keyword to return an iterator that may be iterated over, one value at a time
- Generators do not store their contents in memory
    - More memory-efficient for large data sets
- Generators can be created using a generator expression
    - Similar syntax to a list comprehension but uses rounded brackets instead of squared.
    - The `yield` keyword controls the flow of a generator function
        - Allows the execution to pause and resume while maintaining its state between iterations
- Generators can be used in loops
    - Use next() function to retrieve the next value in the series.
- Generators can be used to represent infinite streams of data,
    - Example – Fibonacci sequence without storing all the values in memory
- Generators use lazy evaluation,

```
def squares_generator(n):
    yield from (x ** 2 for x in range(1, n + 1))
```

```
unique_words = set(word for word in line.split())
```

# RETURN VS. YIELD

- Return terminates the function's execution and returns a single value
- Yield pauses execution and allows for multiple values
- Return discards the function's state after execution
- Yield preserves the state for resumption

# BENEFITS OF YIELD

- Memory Efficiency
  - Generator functions are memory-efficient when dealing with large sequences.
  - They only generate the next value when needed
- Lazy Evaluation
  - Values are calculated only when required, improving performance for computationally expensive sequences.
- Iterators - Generator functions create iterators

# USE CASES

- Infinite Sequences
  - Generating infinite sequences like prime numbers or Fibonacci numbers where you don't know the size beforehand.
- Large Datasets
  - Working with massive datasets where storing everything in memory is impractical.
- Coroutines
  - Implementing cooperative multitasking where multiple functions can yield control and resume based on events.

# TOKENIZER USING A GENERATOR

```python
import re

def tokenizer(input_file):
    with open(input_file, 'r') as f_input:
        for line in f_input:
            for token in tokenize_line(line):
                yield token
def tokenize_line(line):
    tokens = re.findall(r'\b\w+\b', line)
    return tokens
if __name__ == '__main__':
    input_file = 't2.txt'
    token_generator = tokenizer(input_file)
    for token in token_generator:
        print(token)
```

```python
try:
    while True:
        token = next(token_generator)
        print(token)
except StopIteration:
    pass
```