

Skład grupy:

Piotr Łach

Wojciech Maj

Tytuł Pracy:

"Stworzenie narzędzia do analizy leksykalnej i automatycznego dokumentowania kodu, na przykładzie języka Python"

Cel pracy:

Wytworzenie aplikacji: "Narzędzia do analizy leksykalnej i automatycznego dokumentowania kodu, na przykładzie języka Python". Efektem tej operacji ma być plik zawierający dokumentację pythona. Aplikacja została wytworzona z wykorzystaniem biblioteki standardowej pythona w wersji 3.9 oraz moduł dostarczający implementacje kolejek. Całość jest rozwiązaniem autorskim. Bazuje na wyrażeniach warunkowych, które odpowiednie wysterowują treść komentarza. Plik wynikiowy jest połączeniem źródła kodu oraz wygenerowanych komentarzy.

Pojęcia:

- klasa(słowo kluczowe class) - kontener na zminne oraz fukcje(metody) służące określonym zadaniom charakterystycznym dla posiadanych właściwości
- obiekt - reprezentacja klasy, zmienna mająca typ klasy.
- funkcja(metody) - opis czynności wykonywanych na zmiennych/elementach środowiska(dostęp do konsoli) w celu uzyskania założonego rezultatu
- operatory (+/ - %) - elementy o charakterze funkcyjnym, będącymi symbolową reprezentacją bazowych operacji matematycznych, bądź przeciążeń w ramach klasy.
- wątki(threads) - reprezentacja współbieżnie wykonowanego procesu. Pozwala na zarządzanie zasobami oraz kontrolą nad wykonywaniem całości aplikacji, w celu uzyskania optymalnego czasu działania.
- typy złożone - typy zawierające typy proste/bądź złożone np.: listy.
- typy proste(float,char,int) - typy elementarne.
- zmienne - element tworzony bądź/i wykorzystywany w aplikacji posiadający określony, bądź dedukowany typ
- operator '=' - służy jako operacja przypisania wartości/obiektu, do zmiennej/klasz.
- operator '#' - określenie obszaru komentarzy, niewykorzystywana część aplikacji, jedna linia

- operator ' ""{...}"" ' - określenie obszaru komentarzy, niewykorzystywana część aplikacji, obszar pomiędzy znakami ' "" '
- def - słowo rozpoczynające definiowanie funkcji/metody
- import - funkcja załączająca paczkę/plik/moduł
- _funkcja_ - oznaczenie dla specjalnych funkcji
- return - słowo kluczowe zwracające zmienną, będącą wynikiem działania funkcji/metody.
- {} - oznaczenie na słownik, bądź zestaw
- klucz wartość w słowniku - symbolizowane jako {"klucz":"wartość",...,"klucz":"wartość"}
- () - oznaczenie dla tupli
- tupla - obiekt przechowujące elementy
- [] - lista
- zmienna w liście - [el1,el2...eln]
- zmienne w zestawie - {el1,el2...eln}
- zestaw - zbiór niemodyfikowalnych zmiennych
- : - następna linia może zawierać wcięcie
- pętle - narzędzie iteracji

Zawartość projektu

1. Plik 'Documentation_generator_for_Python.py'

W tym pliku znajduje się kod wykonawczy narzędzia będącego tematem projektu. Poszczególne części kodu zostały przedstawione w sekcji : "Prezentacja działania kodu"

2. Plik 'proto_comm.py'

Plik z rozszerzeniem języka Python poddawany analizie i automatycznemu dokumentowaniu jego zawartości. Poszczególne fragmenty kodu poddawane analizie zostały przedstawione w sekcji: "Prezentacja działania kodu"

3. Plik '[file.py](#)'

Plik z rozszerzeniem języka Python poddawany analizie i automatycznemu dokumentowaniu jego zawartości. Poszczególne fragmenty kodu poddawane analizie zostały przedstawione w sekcji: "Prezentacja działania kodu"

4. Pliki 'proto_documentation[1-2].py'

Pliki będące wynikiem działania aplikacji, która jest tematem projektu.

5. Pliki 'file_documentation_1.py'

Pliki będące wynikiem działania aplikacji, która jest tematem projektu.

Prezentacja działania kodu

```

if line.find('from ') != -1 :
    file_write.put_nowait('\n# Importujemy z biblioteki '+dataarray[1]+' metodę/klasę o nazwie:
else:
    if line.find('import') != -1:
        file_write.put_nowait('\n# importujemy bibliotekę o nazwie: '+dataarray[1]+'\\n')

```

```

# importujemy bibliotekę o nazwie: serial
import serial

# importujemy bibliotekę o nazwie: threading
import threading

# importujemy bibliotekę o nazwie: struct
import struct

# Importujemy z biblioteki proto_file.proto_parser metodę/klasę o nazwie: ProtobufCommParser
from proto_file.proto_parser import ProtobufCommParser

# Importujemy z biblioteki proto_file.ExampleProto_pb2 metodę/klasę o nazwie: DataPacket, DeviceRequest, DeviceResponse
from proto_file.ExampleProto_pb2 import DataPacket, DeviceRequest, DeviceResponse

# importujemy bibliotekę o nazwie: logging
import logging

```

Oto przykład wykonania powyższego kodu, który szuka określonych nazw składniowych 'import' i 'from'.

```

if dataarray[0] == 'class':
    file_write.put_nowait('\n# Rozpoczynamy definicję obiektu/klasy\\n')

```

```

# Rozpoczynamy definicję obiektu/klasy
class ProtobufComm:

```

Przykład zamieszczenie komentarzy odnośnego pojawienia się definicji klasy

```

if dataarray[0] == 'def':
    str_line = '\n# Rozpoczynamy definicje funkcji'
    i = str(line).count("__")
    if "__" in line:
        if i == 2:
            print("licznik: "+str(i))
            str_line += ' magicznej'
            file_write.put_nowait('\n'+str_line+'\n')

if "#" in dataarray[0]:
    file_write.put_nowait('# Komentarz programisty\n')

```

Powyższa funkcja jest odpowiedzialna za dodawanie komentarzy odnośnych definicji funkcji a także funkcji tzw. magicznych (np. `__init__`, `__call__`)

```

# Rozpoczynamy definicje funkcji magicznej
def __init__(self, port: str, baudrate: int):
    self.stm = serial.Serial(port, baudrate)

# Rozpoczynamy definicje funkcji
def read_packet(self) -> bytes:
    # Komentarz programisty
    # First, we fetch 2 bytes of size from internal serial buffer
    msg_size_data = self.stm.read(2)
    # Komentarz programisty
    #print(f"Read packet header: {msg_size_data.hex(' ')}")
    msg_len = self.read_packet_size(msg_size_data)[0]
    # Komentarz programisty
    # Then, we fetch the rest of the message

# Rozpoczynamy definicje bloku warunkowego
    if msg_len > 0:
        msg_data = self.stm.read(msg_len)
    # Komentarz programisty
    #print(f"Read packet data: {msg_data.hex(' ')}")

# Zwracanie z funkcji obiektu
    return msg_data
else:
    print("Empty packet.")

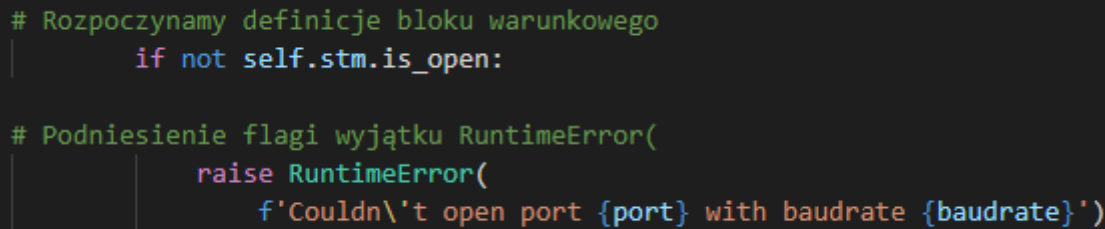
# Zwracanie z funkcji obiektu
    return bytes()

```

Zaprezentowano także dodanie komentarzy generowanych do komentarzy programisty

```
if dataarray[0] == 'if':
    file_write.put_nowait('\n# Rozpoczynamy definicje bloku warunkowego\n')

if "raise" in line:
    file_write.put_nowait('\n# Podniesienie flagi wyjątku '+dataarray[1]+'\\n')
```



```
# Rozpoczynamy definicje bloku warunkowego
|     if not self.stm.is_open:

# Podniesienie flagi wyjątku RuntimeError(
|         raise RuntimeError(
|             f'Couldn\\t open port {port} with baudrate {baudrate}')
```

Przykład dodania komentarza dla bloku warunkowego a także dodanie komentarza dla flagi **raise**

```
if dataarray[0] == 'try:':
    file_write.put_nowait('\n# Rozpoczynamy zagnieżdżenie bloku wyjątku\n')

if dataarray[0] == 'except':
    file_write.put_nowait('\n# Rozpoczynamy zagnieżdżenie wsytąpienie wyjątku: '+dataarray[1]+'\\n')
```

```

# Rozpoczynamy zagnieżdżenie bloku wyjątku
try:
    self.comm = proto_comm.ProtoBufComm( self.port, 115200)
    self.comm.stm.flush()
    self.comm.stm.read_all()
    self.comm.stm.flushInput()
    self.comm.stm.flushOutput()
    self.comm.start_data_output()
    self.com.text += "\nRozpozÅto pomiar\n\n"
    self.com.text = self.comm.start_pom()
    self.csvQueue.put_nowait(self.comm.start_pom())

# Rozpoczynamy pętlę while
while self.trump:
    line_from_stm_output=self.comm.handle_data()
    self.com.text += line_from_stm_output
    self.csvQueue.put_nowait(line_from_stm_output)
    self.comm.stm.flush()
    self.comm.stm.flushInput()
    self.comm.stm.read_all()
    self.comm.stm.flushOutput()
    self.comm.stop_data_output()
    self.comm = None

# Rozpoczynamy zagnieżdżenie wsytąpienie wyjątku: serial.serialutil.SerialException:
except serial.serialutil.SerialException:

# Rozpoczynamy definicję nowego wątku
Thread(target = self.error_win("BŁĄD połączenia się z portem ")).start()

# Rozpoczynamy zagnieżdżenie wsytąpienie wyjątku: struct.error:
except struct.error:

```

Prezentacja przykładu dodania komentarza odnoszącego się definicji zadeklarowania bloku try i except.

```

if "Thread" in dataarray[0]:
    file_write.put_nowait('\n# Rozpoczynamy definicję nowego wątku\n')

if "threading" in line:
    file_write.put_nowait('\n# Rozpoczynamy definicję nowego wątku\n')

if "threading.Timer" in line:
    file_write.put_nowait('# Określenie wątku, który wykona daną metodę w określonym czasie \n')

if ".start()" in dataarray[0]:
    file_write.put_nowait('\n# Startujemy nowy wątek\n')

```

```
# Rozpoczynamy definicje nowego wątku

# Startujemy nowy wątek
|   | Thread(target=self.on_press_false).start()
```

```
if "for" in dataarray[0]:
    file_write.put_nowait('\n# Rozpoczynamy pętle for \n')
```

```
if "while" in dataarray[0] :
    file_write.put_nowait('\n# Rozpoczynamy pętle while\n')
```

```
# Rozpoczynamy pętle while
|   |   | while self.trump:
|   |   |     line_from_stm_output=self.comm.handle_data()
|   |   |     self.com.text += line_from_stm_output
|   |   |     self.csvQueue.put_nowait(line_from_stm_output)
|   |   |     self.comm.stm.flush()
```

```
if "return" in dataarray[0]:
    file_write.put_nowait('\n# Zwracanie z funkcji obiektu\n')
```

```
# Komentarz programisty
|   |   | # using struct to unpack 16-bit unsigned value from byte array

# Zwracanie z funkcji obiektu
|   |   | return struct.unpack('<H', size_data)
```

```
if "logging." in dataarray[0]:
    file_write.put_nowait('\n# Wyświetlenie informacji w konsoli\n')
```

```
# Wyświetlenie informacji w konsoli
|   |   | logging.info(f'{data.valueA}, {data.valueB}, {data.valueC}, {data.constantVa
```

Komentarz odnośnie występowanie tzw. Loggów, które są odpowiedzialne za wypisanie w konsoli informacji, lepiej widocznych niż tych z wykorzystaniem funkcji print.

Dodatkowo można wyświetlić logging.error, który podświetlany jest na czerwono co znacznie pomaga z zauważeniem.