HW 4 Problem 1

Question:

1. Define an enumerated type called TokenType witht the following elements:
    - CONSTANT
    - OPERATOR
    - VARIABLE
    - SPECIAL
2. Define struct/class called Token with the following attributes
    - text
    - token_type
3. Given an input file, create a list of Token objects and print those objects in a custom manner

Provided test cases:

1. Input: “a := 0 + 1;”

```
Token 0 = a
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 0
Token type: constant

Token 3 = +
Token type: operator

Token 4 = 1
Token type: constant

Token 5 = ;
Token type: special symbol
```

2. Input: "b:=1;"

```
Token 0 = b
Token type: variable

Token 1 = :=
Token type: special symbol

Token 2 = 1
Token type: constant

Token 3 = ;
Token type: special symbol
```

Note: First language is Python

Algorithm/Pseudo code:

```
enumerator TokenType:
    CONSTANT
    OPERATOR
    VARIABLE
    SPECIAL

class Token:
    constructor (user_input, id):
        self.text = user_input
        self.id = id
        Assign appropriate token type to self.token_type

    define custom print function

Parse input string
Tokenize each character
print each of them out
```

Actual Code:

```
import enum

class TokenType(enum.Enum):
    CONSTANT = ["0", "1"]
    OPERATOR = "operator"
    VARIABLE = ["a", "b", "c", "d"]
    SPECIAL = [":=", ";"]

class Token:
    def __init__(self, s, id):
        self.text = s
        self.id = id
        if self.text in TokenType.CONSTANT.name:
            self.token_type = "constant"
        elif self.text in TokenType.VARIABLE.name:
            self.token_type = "variable"
        elif self.text in TokenType.SPECIAL.name:
            self.token_type = "special symbol"
        else:
            self.token_type = TokenType.OPERATOR.name

    def __repr__(self):
        print ("Token " + str(self.id) + " = " + self.text + "\nToken
        type: " + self.token_type + "\n")



input = "c:=1 * 1 <= 0* 0 + 0*0;"
input = input.replace(" ", "")
id = 0
i = 0
while i < len(input):
    letter = input[i]
    tok = Token(letter, id)
    print(tok)
    id += 1
    i = i + 1
```

Syntax Error:

- When I defined a custom print function for a Token object in lines 22 and 23, I added a print statement instead of actually returning the string.

Working Code

```
1   import enum
2
3   class TokenType(enum.Enum):
4       CONSTANT = ["0", "1"]
5       OPERATOR = "operator"
6       VARIABLE = ["a", "b", "c", "d"]
7       SPECIAL = [":=", ";"]
8
9   class Token:
10      def __init__(self, s, id):
11          self.text = s
12          self.id = id
13          if self.text in TokenType.CONSTANT.name:
14              self.token_type = "constant"
15          elif self.text in TokenType.VARIABLE.name:
16              self.token_type = "variable"
17          elif self.text in TokenType.SPECIAL.name:
18              self.token_type = "special symbol"
19          else:
20              self.token_type = TokenType.OPERATOR.name
21
22      def __repr__(self):
23          return "Token " + str(self.id) + " = " + self.text + "\nToken
            type: " + self.token_type + "\n"
24
25
26
27  input = "c:=1 * 1 <= 0* 0 + 0*0;"
28  input = input.replace(" ", "")
29  id = 0
30  i = 0
31  while i < len(input):
32      letter = input[i]
33      tok = Token(letter, id)
34      print(tok)
35      id += 1
36      i = i + 1
37
```

Debugging

1. When assigning self.token_type in the constructor of the object, I check if the input is present within the name of the enumerated value instead of list/string associated with it. To fix:

```
def __init__(self, s, id):
    self.text = s
    self.id = id
    if self.text in TokenType.CONSTANT.value:
        self.token_type = "constant"
    elif self.text in TokenType.VARIABLE.value:
        self.token_type = "variable"
    elif self.text in TokenType.SPECIAL.value:
        self.token_type = "special symbol"
    else:
        self.token_type = TokenType.OPERATOR.value
```

2. Did not account for the fact that we can have operators that contain two characters when parsing the input string. To fix:

```
27    input = "c:=1 * 1 <= 0* 0 + 0*0;"
28    input = input.replace(" ", "")
29    id = 0
30    i = 0
31    while i < len(input):
32        letter = input[i]
33        if i < len(input) - 1 and input[i+1] == "=":
34            letter = input[i] + input [i+1]
35            i = i + 1
36        tok = Token(letter, id)
37        print(tok)
38        id += 1
39        i = i + 1
```

3. Made a mistake when doing the second debug step. If there was a “=” after any token, it would be counted as an operator. For example, “0 =” would become “0=”. To fix:

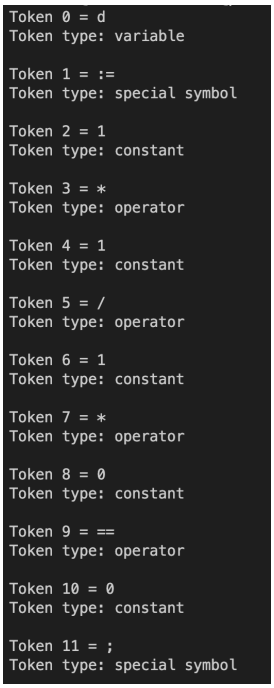
```
38        if i < len(input) - 1 and input[i+1] == "=" and input [i] in ["=", "<", ">", "!", ":"]:
39            letter = input[i] + input [i+1]
40            i = i + 1
```

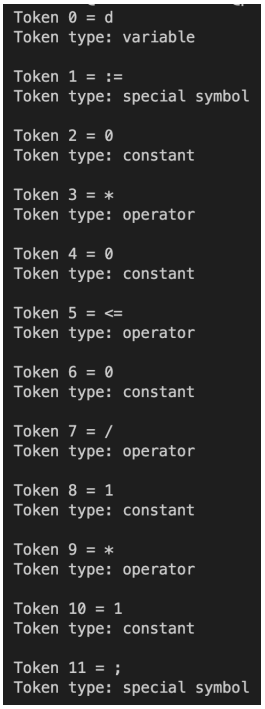
Add Documentation

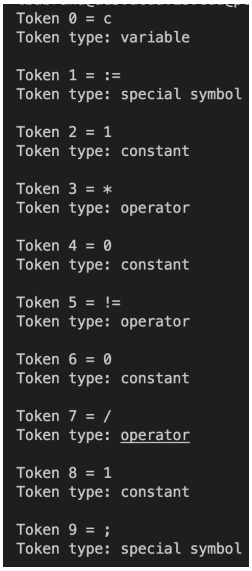
```python
import enum

class TokenType(enum.Enum):
    # Assign enumerated types to their associated values
    CONSTANT = ["0", "1"]
    OPERATOR = "operator"
    VARIABLE = ["a", "b", "c", "d"]
    SPECIAL = [":=", ";"]

class Token:
    def __init__(self, s, id):
        self.text = s   # Assign self.text to input string
        self.id = id    # Assign self.id to user assigned id
        # Compare text to values associated to enums, assigns to matching one
        if self.text in TokenType.CONSTANT.value:
            self.token_type = "constant"
        elif self.text in TokenType.VARIABLE.value:
            self.token_type = "variable"
        elif self.text in TokenType.SPECIAL.value:
            self.token_type = "special symbol"
        else:
            self.token_type = TokenType.OPERATOR.value

    def __repr__(self):
        # Create custom print function for object as required
        return "Token " + str(self.id) + " = " + self.text + "\nToken type: " + self.token_type + "\n"


```

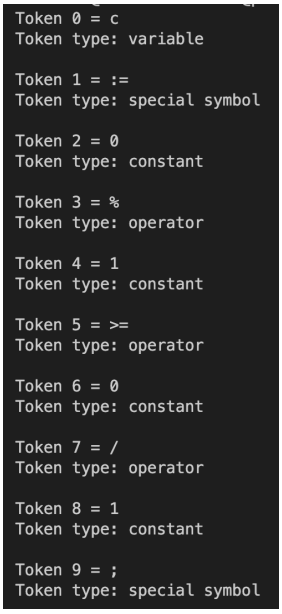
```python
def parse_and_return(inp):
    input = inp.replace(" ", "")
    id = 0
    i = 0
    tokens = []
    # Parsing through input string
    while i < len(input):
        letter = input[i]
        # Accounting for operators with two characters
        if i < len(input) - 1 and input[i+1] == "=" and input [i] in ["=", "<", ">", "!", ":"]:
            letter = input[i] + input [i+1]
            i = i + 1
        # Create Token obj for each token
        tok = Token(letter, id)
        # Append tok obj
        tokens.append(tok)
        id += 1
        i = i + 1

    return tokens

if __name__ == "__main__":
    input = "b:= 1*1;"
    list_of_tokens = parse_and_return(input)
    for tok in list_of_tokens:
        print(tok)
```

Extra Test Cases Used for Debugging:

1. Input: "d:= 1*1 / 1 * 0 == 0;"
   - 

2. Input: "d := 0*0 <= 0/ 1 * 1;"
   - 

3. Input: "c := 1*0 != 0/ 1;"
   - 

4. Input: "c := 0 % 1 >= 0/ 1;"
   - 

Note: Second language is Rust

Actual Code:

```rust
enum TokenType {
    CONSTANT,
    OPERATOR,
    VARIABLE,
    SPECIAL,
}

impl TokenType {
    fn as_str(&self) -> &'static str {
        match self {
            TokenType::CONSTANT => "constant",
            TokenType::OPERATOR => "operator",
            TokenType::VARIABLE => "variable",
            TokenType::SPECIAL => "special symbol"
        }
    }
}

struct Token {
    input: String,
    id: i32,
    text: String
}

```

```
impl Token {
    fn new(u_input: &str, u_id: i32) -> Token {

        let temp_text;
        let constants = vec!["0", "1"];
        let variables = vec!["a", "b", "c", "d"];
        let specials = vec![":=", ";"];
        if constants.contains((u_input)) {
            temp_text = TokenType::CONSTANT.as_str().to_string();
        }
        else if variables.contains((u_input)) {
            temp_text = TokenType::VARIABLE.as_str().to_string();
        }
        else if specials.contains((u_input)) {
            temp_text = TokenType::SPECIAL.as_str().to_string();
        }
        else {
            temp_text = TokenType::OPERATOR.as_str().to_string();
        }
        Token {
            input: u_input.to_string(),
            id: u_id,
            text: temp_text.to_string()
        }
    }

    fn print_self(&self) {
        println!("Token {} = {}", self.input, self.input);
        println!("Token type: {}\n", self.text);
    }
}
```

```
fn main() {
    let mut input:String = "d := 0*0 <=  0/ 1 * 1;".to_string();
    let ops = vec!["=".to_string(), "<".to_string(), ">".to_string(), "!".to_string(), ":".to_string()];
    input = input.replace(" ", "");
    let len:i32 = input.chars().count() as i32;
    let mut  i:i32 = 0;
    let mut id:i32 = 0;
    while i < len {
        let mut letter:String = input.chars().nth(i as usize).unwrap().to_string();
        if i < len - 1 && input.chars().nth((i + 1) as usize).unwrap().to_string() == "=" && ops.contains(&letter) {
            i += 1
        }

        let tok = Token::new(&letter, id);
        tok.print_self();
        id += 1;
        i += 1;

    }

}
```

Syntax Error:

1. In lines 32, 35, and 38, there is an ownership problem with the way that I use u_input. I need to borrow it using its reference.

Working Code

```
enum TokenType {
    CONSTANT,
    OPERATOR,
    VARIABLE,
    SPECIAL,
}

impl TokenType {
    fn as_str(&self) -> &'static str {
        match self {
            TokenType::CONSTANT => "constant",
            TokenType::OPERATOR => "operator",
            TokenType::VARIABLE => "variable",
            TokenType::SPECIAL => "special symbol"
        }
    }
}

struct Token {
    input: String,
    id: i32,
    text: String
}

```

```
impl Token {
    fn new(u_input: &str, u_id: i32) -> Token {

        let temp_text;
        let constants = vec!["0", "1"];
        let variables = vec!["a", "b", "c", "d"];
        let specials = vec![":=", ";"];
        if constants.contains(&(u_input)) {
            temp_text = TokenType::CONSTANT.as_str().to_string();
        }
        else if variables.contains(&(u_input)) {
            temp_text = TokenType::VARIABLE.as_str().to_string();
        }
        else if specials.contains(&(u_input)) {
            temp_text = TokenType::SPECIAL.as_str().to_string();
        }
        else {
            temp_text = TokenType::OPERATOR.as_str().to_string();
        }
        Token {
            input: u_input.to_string(),
            id: u_id,
            text: temp_text.to_string()
        }
    }

    fn print_self(&self) {
        println!("Token {} = {}", self.input, self.input);
        println!("Token type: {}\n", self.text);
    }
}
```

```
58  fn main() {
59      let mut input:String = "d := 0*0 <=  0/ 1 * 1;".to_string();
60      let ops = vec!["=".to_string(), "<".to_string(), ">".to_string(), "!".to_string(), ":".to_string()];
61      input = input.replace(" ", "");
62      let len:i32 = input.chars().count() as i32;
63      let mut  i:i32 = 0;
64      let mut id:i32 = 0;
65      while i < len {
66          let mut letter:String = input.chars().nth(i as usize).unwrap().to_string();
67          if i < len - 1 && input.chars().nth((i + 1) as usize).unwrap().to_string() == "=" && ops.contains(&letter) {
68              i += 1
69          }
70
71          let tok = Token::new(&letter, id);
72          tok.print_self();
73          id += 1;
74          i += 1;
75
76      }
77
78  }
```

Debugging

1. In line 52, self.input is printed twice, instead of self.id being printed once and self.input being printed once. To fix:

```
51      fn print_self(&self) {
52          println!("Token {} = {}", self.id, self.input);
53          println!("Token type: {}\n", self.text);
54      }
```

2. When I encountered a token that was two characters long, I did not concatenate the second character to the first. To fix:

```
67          if i < len - 1 && input.chars().nth((i + 1) as usize).unwrap().to_string() == "=" && ops.contains(&letter) {
68              let b_letter:&str = "=";
69              letter.push_str(b_letter);
70              i += 1
71          }
```

Add Documentation

```rust
// define the enums
enum TokenType {
    CONSTANT,
    OPERATOR,
    VARIABLE,
    SPECIAL,
}

// Assign values to the enumerated types
impl TokenType {
    fn as_str(&self) -> &'static str {
        match self {
            TokenType::CONSTANT => "constant",
            TokenType::OPERATOR => "operator",
            TokenType::VARIABLE => "variable",
            TokenType::SPECIAL => "special symbol"
        }
    }
}

struct Token {
    text: String,
    id: i32,
    token_type: String
}

```

```rust
impl Token {
    fn new(u_input: &str, u_id: i32) -> Token {
        // Figure out which token type the input belongs to
        let temp_text;
        let constants = vec!["0", "1"];
        let variables = vec!["a", "b", "c", "d"];
        let specials = vec![":=", ";"];
        // Check if input belongs to constants
        if constants.contains(&(u_input)) {
            temp_text = TokenType::CONSTANT.as_str().to_string();
        }
        // Check if input belongs to variables
        else if variables.contains(&(u_input)) {
            temp_text = TokenType::VARIABLE.as_str().to_string();
        }
        // Check if input belongs to specials
        else if specials.contains(&(u_input)) {
            temp_text = TokenType::SPECIAL.as_str().to_string();
        }
        else {
            temp_text = TokenType::OPERATOR.as_str().to_string();
        }
        Token {
            text: u_input.to_string(),
            id: u_id,   // assign id to user assigned id
            token_type: temp_text.to_string() // assign token_type
        }
    }

    fn print_self(&self) {
        println!("Token {} = {}", self.id, self.text);
        println!("Token type: {}\n", self.token_type);
    }
}

```

```rust
fn parse_and_return(inp: String) -> Vec<Token> {
    let mut tokens = Vec::new();
    let mut input:String = inp.to_string();
    // Potential operators with two chars
    let ops = vec!["=".to_string(), "<".to_string(), ">".to_string(), "!".to_string(), ":".to_string()];
    input = input.replace(" ", "");
    let len:i32 = input.chars().count() as i32;
    let mut  i:i32 = 0;
    let mut id:i32 = 0;
    // Go through each token
    while i < len {
        let mut letter:String = input.chars().nth(i as usize).unwrap().to_string();
        // Account for operators that have two chars
        if i < len - 1 && input.chars().nth((i + 1) as usize).unwrap().to_string() == "=" && ops.contains(&letter) {
            let b_letter:&str = "=";
            letter.push_str(b_letter);
            i += 1
        }
        // Create new Token object
        let tok = Token::new(&letter, id);
        // Append token to tokens vector
        tokens.push(tok);
        id += 1;
        i += 1;

    }

    return tokens;
}


fn main() {
    let inp:String = "b:= 1*1;".to_string();
    let tokens:Vec<Token> = parse_and_return(inp);
    for tok in &tokens {
        tok.print_self();
    }
```